Introducción al lenguaje C

Introducción a la sintaxis del lenguaje C

Aunque cada uno de los programas que podamos escribir sean distintos, todos tienen características en común. Los elementos de un programa en C serán en general los siguientes:

Comentarios
Inclusión de archivos
variables globales
Definición de funciones creadas por el programador utilizadas en main()
main()
{
    variables locales
    flujo de sentencias
}

Veamos en que consiste cada uno:

Comentarios: Se identifican porque van entre diagonales y asterisco. Nos sirve para escribir información que nos referencie al programa pero que no forme parte de él. Por ejemplo especificar que hace el programa, quien lo elaboró, en que fecha, que versión es, etc. Ej. */*este programa calcula el máximo entre dos números*/*

Inclusión de archivos: Consiste en mandar llamar a la o las bibliotecas donde se encuentran definidas las funciones de C (instrucciones) que estamos utilizando en el programa.
En realidad, la inclusión de archivos no forma parte de la estructura propia de un programa sino que pertenece al desarrollo integrado de C. Se incluye aquí para que el alumno no olvide que debe mandar llamar a los archivos donde se encuentran definidas las funciones estándar que va a utilizar.

Variables globales: Antes de realizar alguna operación en el programa, se deben declarar la(s) variable(s) que se utilizarán en el programa. Estas variables se definen para su utilización en cualquiera de las funciones declaradas.

Definición de funciones creadas por el programador: se define el contenido de las funciones utilizadas. Estas contienen los mismos elementos que la función principal main().

main(): En C, todo está constituido en base a funciones. El programa principal no es la excepción. main() es la función principal del programa la cual se delimita con

llaves. Debe siempre existir y es main() quien puede intercambiar información con el sistema operativo y es siempre la primer función que se ejecuta y también la última dentro de un código.

Variables locales: Antes de realizar alguna operación en el programa, se deben declarar la(s) variable(s) que se utilizarán en el programa. Estas variables se definen para la utilización sólo dentro de la función en la cual se creó.

Flujo de sentencias: Es la declaración de todas las instrucciones que conforman el programa.

Tipos de datos y declaraciones

El programa que hicimos en el módulo anterior sólo nos sirve para desplegar un mensaje en pantalla. Lo verdaderamente satisfactorio es hacer programas que procesen información. Para ello, tenemos que utilizar variables. Veamos como se manejan.
Los tipos de datos son identificadores que usa cada lenguaje para saber la clase de información que va a tomar (manejar) una variable (o una constante también). Por ejemplo: si quiero utilizar la variable "cantidad" para almacenar un número entero, declararé "cantidad" como una variable de tipo **int** lo que significa que sólo aceptará valores de tipo entero.

Los tipos de datos básicos que maneja C son:

char identifica el contenido de la variable que se declare como caracter. Su longitud es de 1 byte.

int se refiere a valores de tipo entero. Ocupa 2 bytes o 4 bytes hecho que depende del contexto esto es si el operativo (y también del compilador) es de 16 bits, 32 bits o 64 bits y para evitar la ambigüedad se utilizan modificadores.

float indica que la variable recibe datos de tipo real con aproximadamente 6 dígitos de precisión. Su longitud es de 4 bytes.

double define variables que aceptan datos de tipo real con 12 dígitos de precisión.
Cada variable ocupa 8 bytes de memoria.

void Este tipo de dato sirve para declarar funciones que no devuelven valores.
Su cantidad de bytes es cero.

Además de los anteriores, podemos crear otros tipos de datos al combinarlos con **modificadores de tipo** como **signed, unsigned, long** y **short**.

También existen los **modificadores de acceso** que sirven para controlar las formas en que se acceden o se modifican las variables. Estos modificadores son **const, volatile** y **static**.

Una variable de tipo **const** trabaja como una constante ya que recibe sus valores por una inicialización explícita o bien por algún medio dependiente del hardware.

Por ejemplo:
const int x;

crea la variable entera x que no puede ser modificada por el programa pero si inicializada.

El modificador **volatile** se usa para indicar al compilador que el valor de una variable se puede cambiar por medios externos (no especificados) al programa. Por ejemplo la dirección de una variable global puede ser pasada a la rutina del reloj del sistema operativo y usada para mantener el tiempo real del sistema. En este caso, el contenido de la variable es cambiado sin que haya una sentencia que así lo indique.
Es importante que respetemos estos tipos de modificadores ya que ayudan a la optimización de las funciones del compilador.
Se pueden usar ambos modificadores juntos.

El modificador **static** se utiliza para que, cada vez que sea llamada una función, no se vuelva a inicializar la variable y mantenga el contenido de la llamada anterior. Por ejemplo, si queremos hacer una función que cuente (1, 2 ,3 ,4 etc.) si no tenemos la precaución que nuestra variable contador sea de tipo **static** siempre que sea llamada la función comenzará desde cero y esto no es lo que queremos.

Otros tipos de datos y modificadores

Muchos tipos de datos son altamente dependientes del contexto (hardware) y del compilador en cuestión. El uso cada vez más extendido de microcontroladores (AVR, PIC, Cortex, etc.) ha generado la necesidad de tener tipos de datos y modificadores de uso específico en un determinado contexto. Por ejemplo existe el tipo de dato **bit**. Si una variable es de tipo bit solo puede tomar dos valores: 0 ó 1. Por otra parte en un microcontrolador, no así en la PC, tendremos modificadores que indicaran en que tipo de memoria será creada la variable o constante que deseemos. Como los microcontroladores en general poseen 3 tipos de memoria distinta, flash rom, ram y eeprom, cada una con sus cualidades propias, se tiene por ejemplo los siguientes casos:

1. int char letra=’A’; /*se almacena en ram y es de lectura-escritura*/
2. flash char letra=’A’; /*se almacena en flash rom y es solo lectura*/
3. eeprom char letra=’A’/*se almacena en eeprom y es de lectura-escritura*/

El caso 1 y 3 es similar salvo que la información que se almacena en ram cuando la maquina no tiene alimentación se borra (se pierde) mientras que si está en eeprom se mantiene. La otra diferencia es que la eeprom es mucho más lenta que la ram en la mayoría de los casos.

La tabla que veremos a continuación se corresponde con los tipos de datos del ANSI C

TIPO en C	TAMAÑO	TIPO	RANGO DE VALORES	
			signed	unsigned
char	1 byte	enteros	-128 a 127	0 a 255
int	2 bytes	enteros	-32.768 a 32.767	0 a 65.535
short int	2 bytes	enteros	-32.768 a 32.767	0 a 65.535
long int	4 bytes	enteros	-2.147.483.648 a 2.147.483.647	0 a 4.294.967.295
float	4 bytes	reales	±(3,4E-38 a 3,4E+38)	NO EXISTE
double	8 bytes	reales	±(1,7E-308 a 1,7E+308)	NO EXISTE
long double	10 bytes	reales	±(3,4E-4932 a 1,1E+4932)	NO EXISTE

A partir del acuerdo 99 la tabla es la siguiente				
TIPO en C	**TAMAÑO**	**TIPO**	**RANGO DE VALORES**	
			signed	**unsigned**
char	**1 byte**	**enteros**	**-128 a 127**	**0 a 255**
int	**Definido por el entorno**	**enteros**		
short int	**2 bytes**	**enteros**	**-32.768 a 32.767**	**0 a 65.535**
long int	**Definido por el entorno**	**enteros**		
float	**4 bytes**	**reales**	**±(3,4E-38 a 3,4E+38)**	**NO EXISTE**
double	**8 bytes**	**reales**	**±(1,7E-308 a 1,7E+308)**	**NO EXISTE**
long double	**Definido por el entorno**	**reales**		**NO EXISTE**

En todos los casos para saber el tamaño se puede utilizar el operador sizeof el cual retorna la cantidad de bytes de la variable en cuestion.

Para el caso de las enteras si la variable es signada los límites son $-(2^{n-1})$ a $2^{n-1}-1$ siendo n la cantidad de bits.
Para variables no signadas los alcances son 0 a 2^n-1 donde n es la cantidad de bits.

Declaración de variables

Después de conocer los tipos de datos existentes en C, veamos como se declaran las variables:
Forma general:

```
tipo_dato
variable;
```

tipo_dato puede ser cualquiera que esté permitido en C y **variable** una palabra que designemos para recibir datos.

No hay cantidad máxima de caracteres (letras o palabras) que podamos utilizar para crear variables pero si hay dos que tengan los primeros 6 caracteres significativos iguales, C las tomará como si fuesen la misma.
Tampoco podemos utilizar como variables aquellos términos que ya han sido definidos previamente por el propio C o por el usuario como variables de otro tipo.

Sentencias de asignación

La forma general de asignar valores a una variable es:

nombre_variable = expresión;

expresión puede ser desde una constante hasta una combinación de variables, operadores y constantes.

Si se mezclasen variables de un tipo con las de otro, se aplica la regla de conversión de tipos que consiste en que el valor del lado derecho de la asignación se convierte al tipo de dato de la variable del lado izquierdo (que es la que recibirá el dato) siempre y cuando ésta última tenga mayor longitud en bytes. Por ejemplo, una variable int podemos asignarla a otra variable de tipo float pero no viceversa.

Inicialización de variables

Inicializar una variable es el darle un valor después que se ha declarado pero antes de que se ejecuten las sentencias en las que se emplea.
En C, se les puede dar un valor a la vez que se declaran. Unicamente se coloca el signo igual y una constante después del nombre de la variable.
La forma general de inicialización es:

tipo nombre_variable =

Ejemplo

```
short uno=1;
float pi=3.1416;
char respuesta='s';
```

Regularmente, las variables locales se inicializan cada vez que se entra en el bloque en el que están definidas mientras que las globales son inicializadas al principio del programa.

Constantes

Constantes son los valores que no pueden ser modificados. En C, pueden ser de cualquier tipo de datos.
Además de los ejemplificados anteriormente, podemos crear constantes de caracteres con barra invertida. Estos corresponden a los caracteres que son imposibles introducir desde el teclado.

+	**suma**
-	**resta**
*	**multiplicación**

Se usan de la misma forma que los caracteres normales por ejemplo:

```
printf('Esta es una prueba\n");
         ch='\0';
```

Operadores aritméticos, relacionales y lógicos

Operador Cast

Un operador es un símbolo que indica al compilador que realice manipulaciones lógicas o matemáticas específicas.

Operadores Aritméticos

Cuando se aplica el operador / a un int o a un char, cualquier residuo se trunca. Por ejemplo 7/3 será 2 en división entera. En cambio 7%2 será 1.
El menos por delante de la variable, hace el efecto de multiplicar su único operando por -1 es decir, cualquier número precedido por un signo menos cambia de signo.

I M P O R T A N T E :

El operador residuo está reservado para variables del tipo entera y devuelve el resto de una división.
Para el caso de necesitar obtener el residuo de variables del tipo reales (definidas como float en lenguaje C), se utiliza la función **fmod(x,y)** presente en la librería **math.h.**

Incremento y decremento

Dos operadores característicos de C son el incremento y el decremento.++ aumenta en uno a su operando y -- le resta 1. Es decir,

x=x+1 es esquivalente a ++x
x=x-1 equivale a - -x

Los operandos de incremento y decremento pueden ir antes o después del operador sin embargo existe una diferencia. Veamos el siguiente ejemplo:

x=10;

Los compiladores de C cambian en forma automática el tipo de una variable en aquellas operaciones en las que intervienen variables de distintos tipos.

>=	**Mayor o igual que**
<	**Menor que**

y=x++; Aquí y tomará el valor de 10 y se denomina postincremento.
y=++x; En este caso, y tomará el valor de 11 y se denomina preincremento.

Los operadores del mismo nivel de precedencia son evaluados por el compilador de izquierda a derecha. Por supuesto, se puede utilizar paréntesis para ordenar la evaluación.
Los paréntesis en C son tratados igualmente que en otro tipo de lenguajes de computadora; se fuerza a que una operación o un conjunto de operaciones tengan un nivel de precedencia mayor.
El lenguaje C admite abreviaturas que simplifican la escritura de ciertos tipos de sentencias de asignación. Por ejemplo:
x=x+10; es igual a x+=10

También, conviene utilizar paréntesis para hacer más claro el orden en que se producen las evaluaciones, tanto para la persona que lo elabora o para los que después tengan que seguir el programa.

Operadores relacionales y lógicos

Los operadores relacionales se utilizan para establecer una relación entre los valores de variables. Devolverán el valor **1** en caso de evaluarse la condición como **VERDADERA** o **0** en caso de resultar **FALSA**.

Como ejemplo, si tenemos dos variables: **cant_naranjas** y **cant_peras,** que representan a la cantidad de naranjas y a la cantidad de peras que hay en un cajón, podremos comparar ambas cantidades por medio de los **operadores relacionales.**
A continuación, la lista de operadores relacionales y su significado:

Los **operadores lógicos** devuelven el valor según sea el resultado de la función lógica aplicada.
Estos son:

Conversiones de tipos de datos

Veamos un ejemplo:

```
float dividendo, divisor;
short cociente_entero;

dividendo = 7.0;
divisor = 2.0;
cociente_entero = dividendo/divisor;
```

Aquí, se han declarado dos variables del tipo **float** (dividendo, divisor) y una del tipo **short** (cociente_entero). Luego de hacerse las asignaciones a las variables que tienen una representación en coma flotante, realizamos la división que nos da el valor de **3.5**, pero no se puede asignar este resultado a **cociente_entero** debido a que ésta es del tipo **short**. Por lo que el valor 3.5 se transforma en un entero, y esto ocurre truncando la parte fraccional. De esta manera, el compilador termina asignando un valor entero (en este caso **3**) a **cociente_entero**.

I M P O R T A N T E :

Observar que realiza un truncamiento de la parte fraccional, y no un redondeo del número.

Operaciones entre tipos de datos distintos

En "C" se pueden realizar operaciones utilizando en las mismas mezclas de tipos de datos. Se verá a continuación como esto influye en el resultado de la operación.

Dado

int a=10, b=3, c;
float x=20, y=7, z;

se verá que se obtiene como resultado de las siguientes divisiones:

c = a / b	**c = 3**
z = a / b	**z = 3.0**
c = x / y	**c =2**
z = x / y	**z = 2,857143**
c = a / y	**c = 1**
z = a / y	**z = 1,428571**

Se puede observar que en el primer caso la división es totalmente entera, perdiéndose los decimales en la operación de división y en su almacenamiento en la variable c.
En el segundo caso sucede lo mismo, pero esta vez se almacena en una variable de tipo float, convirtiéndose el resultado entero a float para su almacenamiento.
En el tercer caso la división es flotante pero se pierden los decimales en su almacenamiento en la variable c.
En el cuarto caso la división y el almacenamiento del resultado es de tipo float por lo que el resultado que se almacena es el correcto.
En el quinto y sexto caso existe una combinación de tipos en la operación de división, convirtiéndose previamente la variable a de entera a flotante y luego realizándose la operación.

Es decir que cuando se combinan varios tipos, la operación se realiza en el formato más grande. En el caso en que el resultado se almacena en una variable de tipo int se pierden los decimales, dando en el último caso el resultado correcto.

operador cast

A veces, si bien los compiladores realizan la conversión explicada, es conveniente sobre todo cuando nuestros programas puedan llegar a ser leídos por otros programadores, realizar la conversión del tipo de una variable en forma explícita.
Se colocará delante de la variable que deseamos modificarle **temporalmente** el tipo, entre paréntesis el tipo que tendrá en esa instrucción.
Ejemplo:

```
short variable_entera1, variable_entera2;
float resultado_flotante;

resultado_flotante = (float)variable_entera1 / variable_entera2;
resultado_flotante = variable_entera1 / (float)variable_entera2;
```

Ambas instrucciones realizan lo mismo, una división entre números flotantes. Cabe destacar que, debido a las reglas del modo aritmético mixto, al ser alguna de las variables del tipo float, se hace una división del tipo float.

Así, podemos decir que el operador cast es un procedimiento utilizado para realizar conversiones de tipo de dato, y que queden explícitas en el código.

Como comentario general, *no es un procedimiento saludable el cambiar la naturaleza de la información.* ¿Qué significa esto en la práctica?: Si un dato o conjunto de datos *son bytes*, no se lo deberá almacenar permanentemente como float, ya que, además de aumentarse la cantidad de memoria utilizada para su almacenamiento inútilmente, hacerse más lento el acceso a la información por mayor cantidad de bytes (4 en lugar de 1) se perderá precisión (en lugar de almacenarse el valor entero 3 tal vez tengamos 3.000002 ó 2.999991 que no es lo que se debía tener!!). Además el manejo de tipos de datos no enteros en muchas situaciones necesita código adicional para su procesamiento, lo que aumenta aun más la cantidad de memoria utilizada en forma innecesaria y haciendo más lento todos los procesos.

Funciones de entrada y salida

getchar()

```
#include<stdio.h>
int getchar()
```

Descripción
Esta función lee un caracter desde el teclado hasta que se pulse <ENTER>.En caso de que se digite más de un caracter antes de pulsar <ENTER>, la variable sólo almacenará el primer caracter que se tecleó.

putchar()

```
#include<stdio.h>
int putchar(ch)
```

Descripción
Escribe un caracter en pantalla donde **ch** puede ser una variable de tipo caracter o un caracter ASCII entre comillas simples(') y posiciona el cursor en la siguiente línea.

kbhit()

```
#include<stdio.h>
int kbhit()
```

Descripción
Esta función no está definida por el ANSI C. Sin embargo, la incluimos aquí porque aunque con nombre diferente, se encuentra definida en todas las implementaciones de C.
Su uso principal es permitir que el usuario pueda interrumpir alguna rutina desde el teclado.
Regresa un valor distinto de cero, si se ha pulsado una tecla y en caso contrario, el valor retornado es cero.

printf()

```
#include<stdio.h>
int printf(formato, lista_arg)
```

Descripción
La función **printf()** despliega en pantalla tanto cadenas constantes (mensajes) como variables de acuerdo al contenido de **formato**.
Con **formato**, se especifica la cadena constante y/o el tipo de variables que desplegará en pantalla. Todo esto siempre va entre comillas("").
Por ejemplo:

```
printf("Hola, como estas?");
printf("%s",nombre);
```

Con la primera instrucción, mandamos un mensaje a pantalla. En este caso, no es necesario especificar algún formato ya que sólo se despliega el mensaje.

Formatos

%c	**Un sólo carácter**
%d	**Decimal**
%i	**Decimal**
%e	**Notación científica**
%f	**Coma flotante**
%g	**utiliza el más corto de %e o %f**
%o	**en sistema octal**
%s	**cadena de caracteres**
%u	**decimal sin signo**
%x	**en sistema hexadecimal**
%%	**Imprimir el símbolo %**
%p	**Presentar un puntero**
%n	**El argumento asociado será un puntero entero en el que se sitúa el número de caracteres escritos hasta entonces.**

Es en el segundo ejemplo de **printf()** donde utilizamos formato para determinar el tipo de variable(s) que van a ser desplegadas. En el caso anterior, determinamos que se va a escribir el contenido de nombre que es una variable de tipo cadena.

Veamos el formato que puede utilizarse para definir el contenido de cualquier variable:

También podemos desplegar en el monitor mensajes como el contenido de variables con el mismo comando **printf()** como en el ejemplo siguiente:

```
printf ("Hola soy %s, tengo %i años.", nombre, edad);
```

el resultado será el siguiente mensaje en pantalla:

<Hola soy Alberto, tengo 35 años>.

Recuerde que se deben especificar en el mismo orden tanto el contenido de las variables a imprimir dentro de formato como las variables en sí.
La función **printf()** devuelve el número de caracteres realmente presentados en pantalla. Un valor negativo significa que se ha producido un error.

Se pueden escribir enteros entre el signo de porcentaje y el caracter que especifica el tipo de dato a presentar. Esto sirve para determinar la longitud del campo, el número de decimales y un indicador de justificación a la izquierda.

Para especificar la longitud del campo, basta con escribir el número después del signo de porcentaje; después se agrega un punto y el número de posiciones decimales que se desea presentar en el caso de los números en coma flotante. Si la cadena es mayor que la anchura del campo, se truncan los caracteres por el final.

Por ejemplo,

%12.4f determina a un número de como máximo doce caracteres de longitud con cuatro posiciones para la parte decimal.

Cuando se aplica a cadenas de caracteres o enteros, el número después del punto determina la máxima longitud del campo. Por ejemplo %3.5s presenta una cadena que tiene al menos tres caracteres y que no excede de cinco. Si la cadena es mayor que el campo, se truncan los caracteres por el final.

Por defecto, toda salida está justificada por la derecha. En otras palabras, si la anchura del campo es mayor que la de los datos presentados, estos son situados en la parte derecha del campo. Puede forzar que la información quede justificada a la izquierda situando un signo menos inmediatamente después del signo de porcentaje. Por ejemplo %-6.3fjustifica un número en coma flotante por la izquierda con tres posicionesdecimales en un campo de seis caracteres.

Ejemplo:

presentará en la pantalla:

hola

scanf()

```
#include <stdio.h>
int scanf (formato,lista_arg )
```

Descripción

Esta función realiza la operación contraria a **printf()**es decir, leé datos de cualquier tipo desde el teclado hasta que se pulse un retorno de carro (<ENTER>).
Sus parámetros también son similares a **printf()** ya que en formato se especifica el o los tipos de variables que se van a leer mientras que en **lista_arg** se escriben las variables.
Por ejemplo:

```
scanf ("%d",&edad);
scanf ("%i%c",&edad,&sexo);
```

La sección de formato corresponde a "**%d**" donde se indica que se va a leer un entero decimal; **&edad** corresponde a **lista_arg** e indica que los caracteres leídos serán almacenados en la dirección que ocupa la variable edad.

Código	Interpretación
%c	**Leer un único carácter**
%d	**Leer un entero decimal**
%i	**Leer un entero decimal**
%e	**Leer un número en coma flotante**
%f	**Leer un número en coma flotante**
%h	**Leer un entero corto**
%o	**Leer un número octal**
%s	**Leer una cadena de caracteres**
%x	**Leer un número hexadecimal**
%n	**Recibir un valor entero igual al número de caracteres leídos hasta entonces.**

La diferencia entre la sintaxis de **scanf()** y **printf()** consiste en que en la lista de argumentos, **scanf()** necesita que se le especifique que el lugar donde vá a almacenar los datos es en la dirección de la variable(**&edad**). La única excepción es cuando se vá a leer una cadena de caracteres ya que este tipo de variables indican una dirección por si mismas.

Ejemplo:

```
char nombre[10];
scanf("%s", nombre);
```

A continuación se presentan los códigos de formato de **scanf()**.

Un espacio en blanco en la cadena de control dá lugar a que **scanf()** salte uno o más espacios en blanco en el flujo de entrada. Un caracter blanco es un espacio, un tabulador o caracter de nueva línea. Esencialmente, un caracter espacio en blanco en una cadena de control dá lugar a que **scanf()** lea, pero no guarde cualquier número (incluido0) de espacios en blanco hasta el primer caracter no blanco.
Un caracter que no sea espacio en blanco hace que **scanf()** lea y elimine el caracter asociado. Por ejemplo, **%d, %d** dá lugar a que **scanf()** lea primero un entero, entonces lea y descarte la coma, y finalmente lea otro número. Si el caracter especificado no se encuentra, **scanf()** termina.
Un ***** situado después del **%** y antes del código de formato leé los datos de tipo especificado pero elimina su asignación. Así, dada la entrada 10/20, el código

```
scanf("%d*%d",&x,&y);
```

asigna el valor 10 a **x**, descarta el signo de división, **y** dá a **y** el valor 20.
Las órdenes de formato pueden especificar un modificador de máxima longitud de campo. Situado entre el **%** y el código de orden de formato, es un entero que limita la cantidad de caracteres a leer para cualquier campo. Por ejemplo, si se quieren leer sólo 15 caracteres en nombre se escribiría así:

Controles de salida

\a	**Señal audible**
\b	**Blanco**
\f	**Pie de página**
\n	**Fin de línea**
\r	**Retorno de carro**
\t	**Tabulador horizontal**
\v	**Tabulador vertical**
****	**Barra invertida**
\'	**Comilla simple**
\"	**Doble comilla**
\?	**Interrogación**
\O	**O = cadena octal**
\xH	**H = cadena**

```
scanf("%15s",direccion);
```

Si el flujo de entrada fuera mayor de 15 caracteres, entonces una posterior llamada de entrada debería comenzar donde esta llamada la dejó. Por ejemplo, si

Av._Corregidora_#_500

se ha introducido como respuesta al **scanf()** anterior, únicamente los primeros 15 caracteres (hasta la a de corregidora) serían situados en dirección debido al especificador de tamaño máximo.
Cuando se hiciera otra llamada a **scanf()** tal como

```
scanf("%s",cadena);
```

_#_500 se asignará a cadena.

Cuando una cadena está siendo leída, los espacios, los tabuladores y los saltos de línea son utilizados como separadores de campos; sin embargo, cuando se leé un único caracter, estos son leídos como cualquier otro caracter.

Bases numéricas y tipos de datos

Como hemos visto anteriormente, existen distintas bases numéricas de uso frecuente en informática: binaria, octal, decimal y hexadecimal.

Dentro del lenguaje C, se pueden utilizar todas ellas.

Si queremos asignar el valor 1245_{10} a una variable entera, haremos lo siguiente:

int A=1245;

Ahora si la magnitud es 1245_8 haremos:

int A=01245;

y el rango de valores en octal será:

0 a 077777	declarada como int
0100000 a 0177777	declarada como unsigned int
0200000 a 017777777777	declarada como long int
020000000000 a 037777777777	declarada como unsigned long int

Si en cambio tenemos 1245_{16} tendremos que proceder así:

int A=0x1245;

y el rango de valores en hexadecimal será:

0X0 a 0x7FFF	declarada como int
0X8000 a 0XFFFF	declarada como unsigned int
0X10000 a 0XFFFFFFF	declarada como long int
0X8000000 a 0XFFFFFFFF	declarada como unsigned long int

Existen compiladores que también pueden manejar la base binaria de la siguiente manera:

int A=0b0010011100110011;

Como se observa, es muy fácil equivocarse a la hora de ingresar una magnitud en formato binario por lo que siempre se prefiere hacerlo en hexadecimal.

NOTA: Los rangos de valores descriptos dependen en general del hardware, del sistema operativo y del compilado en cuestión, por lo que se dan a título de referencia..

El modo texto o consola

Todas las funciones de entrada y salida vistas anteriormente, se aplican pura y exclusivamente a lo que se suele denominar modo consola o modo texto. En este modo la información siempre termina siendo ASCII, esto es, si queremos escribir el número 18476 en la consola este será

“descompuesto” en componentes ASCII y por lo tanto lo que veremos será el caracter 1, el caracter 8, el caracter 4, el caracter 7 y el caracter 6. En definitiva, se produjo un cambio de formato para adaptarse al elemento en cuestión. El número 18476 estaba almacenado en memoria como 2 bytes (int n=18476) y fué convertido por la función printf a 5 bytes cuyo contenido es: 49, 56, 52, 55, 54. Es decir que lo convirtió a un vector de enteros de 5 bytes que corresponderán a cada una de las posibles posiciones de nuestra “pantalla”. En verdad nuestra pantalla puede ser vista como una matriz de caracteres ASCII de 80x25, que es el modo en el que arranca la PC. Pero ojo, para el ser humano la pantalla es una matriz (dos dimensiones), pero para el hardware siempre será un vector (una dimensión). Por lo tanto, printf recibirá datos de los tipos válidos que maneja (float, int, long, etc.) y siempre los convertirá, utilizando la información que le brinda los modificadores (%f, %d, %u, etc.) a una cadena ASCII. En forma inversa se puede decir que el scanf lee una cadena ASCII “desde el teclado” y mediante la información de los modificadores, la convierte a int, short, float, etc. Tenga en cuenta que printf y scanf forman parte de una familia de funciones y no siempre el destino final es la pantalla y la entrada es un teclado.

Capítulo 5

Control de flujo en lenguaje C

Sentencias de control

En este módulo veremos lo relacionado a las sentencias de control que maneja C. Es importante que se comprendan bien porque son herramientas básicas en la implementación de programas.
De acuerdo a la sintaxis utilizada en C, una **sentencia** puede ser una proposición o ninguna proposición (si hablamos de una sentencia vacía) o un conjunto de proposiciones (cuando se encuentran agrupadas entre llaves). Dentro de este capítulo utilizaremos la palabra sentencia con las tres acepciones.

Proposición if-else

La sentencia de control **if** nos sirve para verificar que se cumpla una condición en el programa.
Su forma general es

```
if (expresión)
        sentencia_1;
else
        sentencia_2;
```

sentencia puede ser una sola proposición o un conjunto de proposiciones delimitadas por llaves.

Ejemplo:

Queremos dividir dos números cualquiera donde el divisor sea diferente de cero. Necesitamos dos variables de tipo real (pero que también acepte valores enteros) para recibir los datos y una para almacenarlo.

```
Codificado en C:

if (divisor!=0)
     divisior=dividendo/divisor;
else
     printf("Error en el divisor");
```

Si se cumple la condición (que divisor sea diferente de cero) entonces se realizará la división; si no, se mandará un mensaje de error.

I M P O R T A N T E :

El lenguaje C NO PERMITE ausencia de sentencia/s en la opción verdadera cuando hay sentencia/s en la opción falsa (else). En cambio, sí PERMITE la falta de sentencia/s en la opción falsa (else).

Ifs anidados

La expresión "ifs anidados" se refiere a que podemos utilizar una sentencia **if** dentro de otra sentencia **if**. Esto se emplea cuando se tienen que cumplir varias condiciones para poder ejecutar una acción.
Ejemplo:
Hagamos un algoritmo que seleccione una asignatura.

Codificándolo en C:

```
#include<stdio.h>
menu()
       {
       char opcion;
       printf("A)Español\n");
       printf("B)Matemáticas\n");
       printf("C)Historia\n");
       printf("D)Biología\n");
       printf("Opción: ");
       scanf("%c",&opcion);
       printf("Escogió ");
       if (opcion=='A')
              printf("español");
```

```
switch (expresión)
        {
        case constante1: secuencia de sentencias; break;
        case constante2: secuencia de sentencias; break;
        case constente3: secuencia de sentencias; break;
        .
        .
        .
        default: secuencia de sentencias; break
(opcional)
```

```
        else if (opcion=='B')
              printf("matemáticas");
              else if (opcion=='C')
                    printf("historia");
                    else if (opcion=='D')
                          printf("biología");
                          else printf("una asignatura inexistente.");
}
```

En el programa anterior, después de haber mandado a pantalla las opciones, leemos la respuesta. Para determinar la opción que se escogió, se recurre a las sentencias **if**. Cuando la evaluación sea verdadera es decir, cuando encontremos el valor que leímos en **opción**, ejecutaremos en este caso la función correspondiente.
Las condiciones se codifican en orden de importancia es decir, la condición más importante se evalúa primero, enseguida la que le sigue en importancia y así sucesivamente hasta llegar a la menos importante. En este caso, todas las condiciones eran igualmente importantes por lo que no importa el orden en que se evalúen.

Proposición switch-case

Cuando se anidan muchos **if**s, el código puede volverse difícil de entender aún para el autor. Para darle mayor transparencia, en C utilizamos la sentencia de decisión múltiple **switch**.
switch es utilizada cuando una variable es sucesivamente comparada con una lista de **enteros o caracteres**. Cuando se encuentra la correspondencia, se ejecuta una sentencia o el bloque de sentencias.

La forma general de una sentencia **switch** es

switch es el indicador del tipo de sentencia de control.

Las llaves **{ }** delimitan lo que abarca la sentencia **switch**.

case especifica donde comienza la evaluación de la variable con la constante que sucede al **case**
En caso de que encontremos el valor de la variable en un case se ejecutarán la secuencia de sentencias relacionadas con este hasta encontrar un **break**.
Las sentencias asociadas a **default** serán ejecutadas si no se encuentra ninguna correspondencia con el valor de la variable que estamos evaluando en alguno de los **case**. El final de estas sentencias las determina la **}** del **switch**. Esta sentencia es opcional (se puede o no incluir). Si no está presente, no se ejecutará ninguna sentencia en caso de que ningún **case** corresponda al valor que estamos buscando.

N O T A

No obstante de figurar como opcional el break en el default, Kernighan y Ritchie recomiendan siempre colocarle el break.

I M P O R T A N T E

La diferencia entre switch e if es que la primera sólo puede comprobar la igualdad, mientras que if puede evaluar expresiones relacionales o lógicas.

No puede haber dos constantes **case** que tengan los mismos valores. Sólo en el caso de que la sentencia **switch** este anidada (que haya una dentro de otra).

Ejemplifiquemos el **switch** optimizando el ejemplo anterior. Ahora, codifiquemos el algoritmo de menú, pero utilizando un **switch** en vez de **ifs.**

```
menu()
	{
	char opcion;
	printf("A)Español\n");
	printf("B)Matemáticas\n");
	printf("C)Historia\n");
	printf("D)Biología\n");
	printf("Opción: ");
	scan("%c",&opcion);
	printf("Escogió ");
	switch(opcion)
		{
		case 'A':printf("español");break;
		case 'B':printf("matemáticas");break;
		case 'C': printf("historia");break;
```

```
        case 'D': printf("biología");break;
        default: printf("escogiste una asignatura inexistente");
        }
}
```

Bucles condicionales

Se denomina **bucle, lazo o ciclo** a las proposiciones de control que nos permiten ejecutar una sentencia hasta que se cumpla cierta condición.
Entre sus ventajas se encuentra, el compactar el código es decir, si tenemos que realizar una tarea varias veces, basta con escribirla una sola vez dentro de un bucle, para que pueda efectuarse las veces que deseamos. También ayuda a que el código sea más entendible.
Los bucles soportados por C son el el **while** y el **do-while** y el **for** como caso particular del while.

Proposición while

Este es otro de los bucles disponibles en C.
Su forma general es

```
while (condición)
        {
        sentencia;
        }
```

donde

sentencia puede ser:
- una sentencia vacía
- una única sentencia
- un bloque de sentencias, caso en el cual se hace imprescindible la utilización de llaves.

Las llaves no son necesarias cuando **sentencia** es una sola instrucción, pero se recomienda utilizarlas para mejorar la legibilidad del programa.

El bucle itera mientras se cumpla la condición (puede ser cualquier expresión). Cuando ya no se cumple, el control del programa pasa a la línea siguiente al código del bucle.

Ejemplo:
Hagamos un programa que nos devuelva el valor absoluto de un número y que se repita todas las veces que deseemos. Necesitaremos una variable tipo **float** (para que tenga mayor rango)que reciba el número.
También ocuparemos una variable que nos sirva para determinar si continua o no en el programa. Si la variable es igual a 'S' continuará y si es igual a 'N', terminará. Sólo aceptará esos dos valores.

Codificado en C:

```
#include<ctype.h>
#include<conio.h>
#include<stdio.h>
#include <math.h>

main()
	{
	char respuesta;
	clrscr();
	respuesta='\0';
	while (respuesta!='N')
		{
			float x;
			printf("Dame un número: ");
			scanf("%f",&x);
			x=fabs(x);
			printf("Su valor absoluto es: %f\n",x);
			printf("Presione 'N' para salir...\n");
			respuesta=toupper(getch());
		}
	}
```

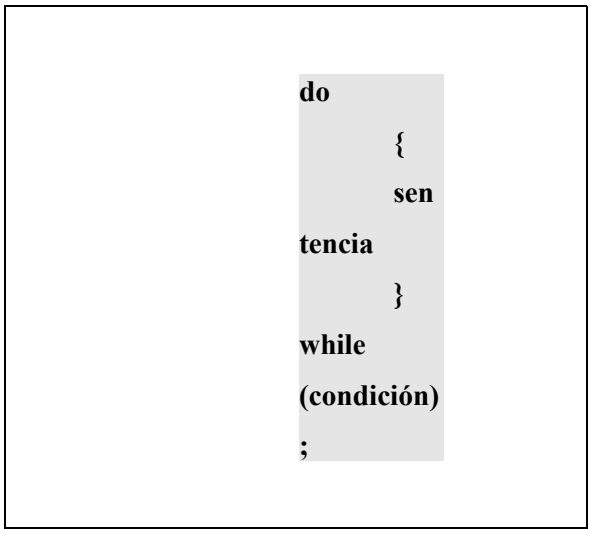
Inicializamos la variable respuesta con caracter nulo **('\0')** para asegurarnos de que al menos se entre una vez al ciclo.
Al final del bucle, preguntamos si quiere continuar o no. Con **getch()** leemos el caracter. **toupper()** nos sirve para convertir el caracter recibido por **getch()** a mayúsculas; de esta manera validamos que se acepte 'S' o 'N' minúsculas.
En caso de que la respuesta sea afirmativa, se repite el ciclo.
Como en el caso del **for**, podemos utilizar bucles **while** sin cuerpo.
Por ejemplo:

while((letra=getch())!='M');

En letra se guardará el valor que se lea con **getch()** y esto se repetirá hasta que letra sea igual a 'M'.

Proposición do-while

La característica principal del **do-while** es que analiza la condición del bucle al final del ciclo, lo que garantiza que el bloque de sentencias se ejecute al menos una vez. Su forma general es



donde **sentencia** puede ser:
- una sentencia vacía
- una única sentencia
- un bloque de sentencias, caso en el cual se hace imprescindible la utilización de llaves.

Las llaves no son necesarias cuando **sentencia** es una sola instrucción, pero se recomienda utilizarlas para mejorar la legibilidad del programa.

Como ejemplo, modifiquemos el programa anterior utilizando **do-while**:

```
#include<ctype.h>
#include<conio.h>
#include<stdio.h>
#include<math.h>

main()
       {
       char respuesta;
       clrscr();
       do
              {
              float x;
              printf("Dame un número: ");
              scanf("%f",&x);
              x=fabs(x);
              printf("Su valor absoluto es: %f\n",x);
              printf("Continuar...\n");
              respuesta=toupper(getch());
              }
       while(respuesta!='N');
       }
```

En esta ocasión, no necesitamos inicializar respuesta ya que por las características de **do-while** de todos modos se ejecutaría la primera vez. La única diferencia es que ahora la condición se comprueba al final del bucle y no al principio como lo hace **while**.

```
for (declaraciones e inicializaciones; condición; incrementos)
        {
        sentencia;
```

Proposición for

El formato general para implementar el bucle **for** es

inicialización es una sentencia de asignación a una variable de control del bucle. En dicha asignación se impone el valor de comienzo.
condición es una expresión que nos indica la condición que debe cumplirse para que continúe el bucle. Implícitamente define el corte del ciclo.
incremento define como va cambiando el valor de la variable de control cada vez que se repite el bucle.
sentencia es el conjunto de proposiciones que se van a realizar en cada uno de los bucles y puede ser:
- una sentencia vacía
- una única sentencia
- un bloque de sentencias, caso en el cual se hace imprescindible la utilización de llaves, si bien las llaves no son necesarias cuando **sentencia** es una sola instrucción, pero se recomienda utilizarlas para mejorar la legibilidad del programa.

Por ejemplo hagamos algoritmo para crear una línea en la pantalla un sencillo marco con código ASCII.

También pueden realizarse varias declaraciones e inicializaciones (deben ir separadas por comas):

Ejemplos de declaraciones for

1)

```
short b;
for (short a=0, b=10; a < 9 ; a++, b--)
      printf ("a= %d, b= %d", a, b)
```

En este caso, estamos inicializando **a** y **b** dentro del bucle. Al mismo tiempo, en la sección de incremento, aumentamos **a** y disminuimos el valor de **b**. Esto es completamente válido.

2)

```
for (scanf("%d", &N),I = 0,S = 0 ; N > = 0 ; S += N,scanf(("%d", &N), I++)
```

El valor N de inicialización de la condición es declarado por un ingreso scanf . También, al finalizar el lazo, **S** acumula los valores de **N**, Iseincrementa y se cambia nuevamente por teclado el valor de N. Completamente válido en el lenguaje C (solamente).

Ejemplos de aplicación

Realizar un programa que convierta un número en sistema decimal al sistema binario.

Para esto, vamos dividiendo al número en base 10 por 2, repitiendo hasta que el cociente de menor que 1. Tomamos los restos de cada división, el último cociente y tenemos el binario buscado.

Diagrama de CHAPIN

Y su codificación en C será:

```
#include<stdio.h>
#include<math.h>

void main (void)
       {
       char resto, potencia = 0;
       short numero;
      unsigned long int BIN = 0;

      printf ("Ingrese el número a convertir al binario\n");
      scanf ("%d", &numero);
      while (numero > 0)
       {
              resto = numero%2;
              numero = numero/2;
              BIN = BIN + resto*pow(10,potencia);
              potencia++;
              }
      printf ("En binario es %d\n", BIN);
       }
```

Programa que dibuja un marco en la pantalla con código ASCII.

El pseudocódigo podría ser

inicio
desde (x=1;x<80;incrementa x)
inicio
posicionarse(x,1);
escribe ("|");
posicionarse(x,24);
escribe("|");
fin
desde(x=1;x<80;incrementa x)
inicio
posicionarse(1,x);
escribe("-")
posicionarse(80,x);
escribe("-");
fin
fin

Codificándolo en C

```
main()
	{
	for(x=1;x<80;x++)
		{
			gotoxy(x,1);
			printf("|");
			gotoxy(x,24);
			printf("|");
		}
	gotoxy(1,1);
	printf("┌");
	gotoxy(80,1);
	printf("┐");
	gotoxy(1,24);
	printf("└");
	gotoxy(80,24);
	printf("┘");
	for(x=0;x<24;x++)
		{
			gotoxy(1,x);
			printf("-");
			gotoxy(80,x);
			printf("-");
		}
	}
```

gotoxy es una función que ubica -mediante parámetros – al cursor en determinada coordenada de la pantalla. Los parámetros se citan como en coordenadas cartesianas, sabiendo que la pantalla posee 80 posiciones en horizontal y 25 (renglones) en vertical.

x=1 nos indica que vamos a empezar a contar desde el 1. Después se verifica que x sea menor que 80 o 24(en el segundo for). Si es así, se ejecutan las sentencias que se encuentran dentro del bucle para finalmente, incrementar el valor de x y volver a evaluar. El ciclo se termina cuando x es igual o mayor a 80 o que 24 (en el segundo for).

```
for ( x=0;  ; x++ )
        printf          ("Bucle
```

Una de las principales características del **for** es que la condición (para determinar si se ejecutan o no las sentencias que contiene) se evalúa al principio del ciclo. Por esto, el código dentro del bucle no será ejecutado si la condición es **false** al comienzo.
Debido a la flexibilidad del lenguaje C, se puede utilizar el operador coma para permitir dos o más variables de control.

Bucle Infinito

Es aquel que no tiene fin. En este tipo de ciclos, no necesitamos especificar ninguna de las tres proposiciones que lo constituyen

o basta dejar la de la condición vacía como se muestra a continuación:

La forma de romper un bucle infinito es mediante la sentencia **break**. Cuando se encuentra esta proposición dentro de un ciclo, trunca el bucle y se procede a ejecutar el código que se encuentra inmediatamente después de él.

```
main()
        {
        for(;;)
                {
                printf("Número: ");
                scanf("%d",&x);
                if(x==9) break;
                }
                printf("Digitaste %d números antes de romper el ciclo",x);
        }
```

El bucle anterior se repite hasta que se teclea el número 9 y después se imprime en pantalla la cantidad de veces que pulsó un número antes de romper el ciclo.

Bucle sin cuerpo (retardos en la ejecución)

Según la sintaxis de C, existen las sentencias vacías por tanto, podemos tener un bucle **for** o cualquier otro, que no tenga cuerpo. Este tipo de ciclos se usan regularmente para retardar la ejecución del programa. Veamos como crear uno:

```
main()
{
short num;
```

```
printf("Piensa un número del 1 al 10.");
lapso();
printf("Que número pensaste: ");
scanf("%d",&num);
.
.
.
}

lapso()
{
short tiempo
short valor=100;
for (tiempo=0;tiempo<valor;tiempo++);
}
```

En el programa anterior, se dá cierto tiempo para que pienses un número y luego lo escribas.

break

La sentencia **break** tiene dos usos:

1) Para determinar el final de un **case** en una sentencia **switch** analizado en este mismo capítulo, y
2) Terminar la ejecución de un bucle y saltar a la sentencia que sigue al ciclo que contiene el break.
Veamos en que consiste el segundo uso:
Cuando utilizamos una sentencia **break** dentro de un bucle, éste se termina inmediatamente y el control del programa pasa a la línea que está inmediatamente después del ciclo.

Veamos gráficamente el corte abrupto del ciclo:

Ejemplo

```
#include<stdio.h>
main()
{
	short x;
	for(x=0;;)
	{
		if(x==250)
			break;
		x++;
	}
	printf("%d",x);
}
```

El programa anterior hay un ciclo infinito que es truncado cuando **x** es igual a **250**.

exit()

Esta sentencia da lugar a la terminación del programa.
Los programadores la utilizan cuando no se satisface una condición obligatoria en la ejecución de un programa ya que detiene su ejecución y fuerza la vuelta al sistema operativo.
La función **exit()** requiere de un argumento entero ya que se supone que un proceso de alto nivel sería capaz de acceder al valor contenido en el argumento. Normalmente se utiliza un argumento 0 (exit(0)) para indicar que se trata de una terminación normal y otros argumentos para indicar algún tipo de error.

Por ejemplo, imagine un programa al que se tenga que dar una clave de acceso para iniciar.
La función **main()** de este programa sería como esta:

```
main()
	{
	char clave[]="clave";
	char password[5];
	printf("Proporcione clave de acceso: ");
	scanf("%s",password);
	if(password!=clave)exit(1);
	.
	.
	.
	}
```

Al iniciar el programa, se pide la clave de acceso. En caso de que no sea la especificada, el programa termina sin esperar nada más.

Continue

Es una sentencia de significado opuesto al **break**. En vez de forzar la terminación del bucle, **obliga** una nueva iteración del ciclo y salta cualquier código que exista desde el continue hasta el final del ciclo. En el caso de un **for**, **continue** realiza los incrementos de la estructura iterativa y pasa el control a la condición del ciclo.

Gráficamente, realiza lo indicado por la flecha:

Por ejemplo:

```
do
       {
       short years,tope=12;
       gotoxy(5,5);
       clreol();
       printf("Años: ");
       scanf("%d",&years);
       if (years<=0) continue;
       if (years<=tope)
       printf("Lo siento, este programa no es para niños");
       }
while(tope!=18);
```

En el ciclo anterior, utilizamos **continue** para verificar que la edad sea mayor que cero. En caso de que ésta sea cero o menos vuelve a comenzarlo sin llegar a ejecutar las instrucciones del bucle restantes. Al no comparar la condición que puede terminar el ciclo, éste se repite.
No es de preocuparse si por ahora no se sabe bien cuando utilizar un while o do-while. La experiencia le irá indicando cuando es más conveniente utilizar uno u otro. Lo más importante es entender el funcionamiento de cada uno.

Funciones en lenguaje C

Definición de una función

En C, una función es un bloque de instrucciones que realizan una tarea específica la cual se maneja como una unidad lógica. Esta unidad, regresa un valor de acuerdo al resultado del proceso que realice.
C es un programa de funciones, todo se hace a partir de ellas. La principal es **main(),** la cual utiliza a otras que se encuentran definidas en las bibliotecas (todas las instrucciones que maneja C).
Además de éstas, nosotros podemos definir nuestras propias funciones. De esta manera dividimos tareas grandes de computación en varias más pequeñas lo que nos dá como resultado que el programa sea más fácil de entender y se pueda manejar más eficientemente.
Además, al subdividir los programas en funciones, éstas pueden ser re-utilizadas en otros programas.
En esta unidad estudiaremos como crear nuestras propias funciones.

La forma general para definir una función es:

```
especificadordetipo  nombrefunción  (declaración de
parámetros)
        {
        cuerpo de la función
        }
```

El **especificadordetipo** de la función define la clase de valor (tipo de dato) que regresa la función. El valor puede ser cualquier que maneje el C.

N O T A

En caso que no se especifique ninguno, la función devolverá por omisión un entero (tipo int).

---

nombrefunción es la palabra con la que vamos a identificara la función.

La **declaración de parámetros** es un conjunto de variables separados por comas y con un tipo de dato específico que reciben los valores de los argumentos cuando se llama a la función.

Una función puede carecer de parámetros en cuyo caso los paréntesis estarán vacíos tanto al declarar como al mandar llamar a la función.
Por ejemplo:

```
float multiplicación(float multiplicando, float multiplicador)
{
	multiplicando=multiplicando*multiplicador;
	return(multiplicando);
}
```

La función está declarada de tipo **float** porque es la clase de valor que va a regresar.
multiplicando y **multiplicador** es el nombre que le vamos a darlos valores que recibirá la función para trabajar con ellos los cuales, son declarados dentro de los paréntesis.
Por último se define el cuerpo de la función (delimitándose con llaves).
En este caso, asignamos el resultado de la multiplicación a multiplicando para ahorrar memoria (se declara una variable menos) y regresamos el valor obtenido por medio de la sentencia **return** (ver más adelante proposición return)**.**

Variables locales y globales
Reglas de existencia

De acuerdo al lugar donde se declaran, las variables pueden ser **locales** o **globales**. Exite un caso particular de las locales que son los parámetros **formales**.

Variables locales

Son aquellas que se declaran dentro de un conjunto de código relacionado lógicamente entre si.

Por ejemplo:

```
funcion_a()
	{
	int x;
	.
	.
	.
	}
funcion_b()
	{
	int x;
	.
	.
	.
	}
```

En las funciones anteriores, declaramos dos variables llamadas "x" con el mismo tipo. Al tratar de utilizar alguna de las variables fuera de las funciones a las que pertenecen, se marcará un error porque el tipo sólo ha sido definido para utilizarse dentro de las llaves donde fue declarada (dentro de ese conjunto relacionado lógicamente).
El que ambas variables se llamen "x" no quiere decir que sean la misma o que ocupen el mismo lugar en la memoria dado que las variables locales se crean y se destruyen cada vez que se sale del bloque en el que son declaradas por tanto, también su contenido se pierde al salir de él.

Parámetros formales

Su comportamiento es el mismo al de cualquier otra variable local.
Muchas de las funciones necesitan argumentos. Los argumentos son datos que recibe la función desde el exterior. A las variables (locales) que sirven para transmitir estos datos se les llama **parámetros formales de la función**. Veamos el siguiente ejemplo:

/*Devuelve la suma de dos números dados*/

```
suma (int a, int b)
{
      a+=b; /*A a toma el valor de a+b*/
      return(a);
}
```

Esta función realiza la suma de dos números. Utiliza dos parámetros formales: **a** y **b** los cuales deben ser declarados en la llamada a **suma()** para luego poder utilizarlas dentro de la función como variables locales. También se destruyen al salir de la función.
Se debe tener mucho cuidado de que los parámetros formales que se declaran sean del mismo tipo que los valores que se introducen cuando se manda llamar la función porque se podría obtener resultados inesperados.

Variables globales

Este tipo de variables se conocen a través de todo el programa y de todas las funciones declaradas en él, por lo cual se pueden usar en cualquier parte de la ejecución de éste. Se declararán antes de cualquier función y se recomienda que sea al principio del programa para mantener un orden.
Consideremos el ejemplo que se presenta a continuación, donde utilizamos los diferentes tipos de variables:

/ Programa que eleva al cuadrado un número real */*

```
#include<math.h>
float cuadrado(float);

float x;

main()
      {
      printf("Ingrese un número: ");
      scanf("%f",&x);
      x=cuadrado(x);
      printf("El cuadrado es %f",x);
      }

float cuadrado(float n)
      {
      n=n*n;
      return(n);
      }
```

Si analizamos las sentencias, vemos que **x** es una variable global mientras que **n** es un parámetro formal que nos sirve como variable local para efectuar la función.

Proposición return

Esta sentencia se utiliza para devolver valores generados en una función a la función que la invoque. También sirve para salir de la función donde se encuentra y continuar con la instrucción posterior a la función que lo llamó.
En la función anterior **return (n);** devuelve el contenido de **n** al programa principal. En caso de que sólo se quiera salir de la función, no es necesario indicarle parámetros, basta con **return()**;.
Todas la funciones, excepto las de tipo **void** (es el tipo de datos que no tiene valor) generan valores que se transmiten al programa principal. Estos valores son de tipo **int** (entero) por omisión, pero puede regresarlos de todo tipo si así se declara; por ejemplo la función **cuadrado**, regresa un valor de tipo **float** (flotante).
Cuando queremos utilizar el valor que devuelve la función en el programa principal, es necesario asignar la función a una variable del tipo de dato que va a regresar la función.

Reglas de las funciones

- No podemos declarar funciones dentro de funciones ya que todas están al mismo nivel.
- Tampoco podemos ingresar al código de una función si estamos fuera de la misma.
- Las variables que se declaran en las funciones, son locales y no pueden ser utilizadas fuera de esa función.

Además, si utilizamos una función dos o más veces, la segunda vez las variables locales no contendrán el valor que obtuvieron al ejecutar la primera vez la función y así sucesivamente ya que se crean al entrar a la función y se destruyen al salir.

Invocación y llamada por valor

Argumentos de las funciones

Es muy común que las funciones utilicen argumentos es decir, que necesiten de algún valor o valores externos dentro de su propio código. Estos valores se pasan mediante variables denominadas parámetros formales de la función las cuales se declaran dentro de los paréntesis que suceden a el nombre de la función o bien, después de estos y antes de la llave de comienzo.
Asegúrese de que de que los argumentos utilizados al declarar la función sean del mismo tipo que los usados para llamar la función. Si hay algún error en los tipos, el compilador no mandará mensaje de error pero se obtendrán resultados inesperados. En algún caso se obtendrán *warnings*.
Se puede utilizar a las variables que son parámetros formales como cualquier otra variable local es decir, se les puede hacer asignaciones o usarlos en cualquier expresión permitida por C.
Existen dos formas de pasar argumentos a una función: por **valor** o por **referencia**
La primera es por medio de las:

Llamadas por valor

Consiste en sólo pasar el contenido de la variable utilizada como argumento a la subrutina. De esta manera, los cambios efectuados en los parámetros de la función no afectan a las variables (globales) que se utilizaron para hacer la llamada a la función.

Llamadas por referencia

Se pasa a la subrutina la **dirección** de la variable que se está mandando como parámetro. De esta manera, los cambios que sufra el parámetro dentro de la subrutina, se efectuarán también en la variable que se introdujo como parámetro.
La forma de pasar una llamada por referencia es pasando un puntero al argumento, de esta manera, lo que pasará es la dirección de la variable en vez de su contenido. Para esto, los parámetros se declaran de tipo puntero.
Cabe aclarar que estamos introduciendo el concepto de puntero que no es objetivo del presente texto. La exposición de las llamadas por referencia es al simple hecho de nombrarlas. El lector podrá ahondar estos conceptos en bibliografía que trate el tema punteros.

```
/*Ejemplo de funciones llamadas por valor y por referencia
//Calcula dos veces el porcentaje de gastos, la primera vez utilizando una función
con //llamada por valor y la segunda por referencia*/

#include<stdio.h>

porcentaje_xvalor(float ingreso, float egreso)
	{
	egreso=((egreso/ingreso)*100)
	printf("Usted gasta el %.2f por ciento de lo que gana",egreso);
	}

porcentaje_xref(float *ingreso, float *egreso)
	{
	*egreso=(((*egreso)/(*ingreso))*100);
	printf("Usted gasta el %.2f por ciento de lo que gana",egreso);
	}

main()
	{
	float entrada,salida;
	clrscr();
	printf("Entradas: ");
	scanf("%f",&entrada);
	printf("Salidad: ");
	scanf("%f",&salida);
	porcentaje_xvalor(entrada,salida); /*Llamada a la función porcentaje utilizando
	paso de parámetros por valor*/
	printf("\n\n");
	porcentaje_xref(&entrada,&salida); /*Utilización de la función porcentaje con paso
	deparámetros por referencia*/
	getch();
	}
```

En el programa anterior, realizamos la misma tarea dos veces pero de diferente manera.

En **porcentaje_xvalor(entrada,salida)** mandamos el contenido de **entrada** y **salida** a la función donde son recibidos por **ingreso** y **egreso** respectivamente. De esta manera, el cálculo del porcentaje se hace internamente es decir, utilizando las variables definidas en la función. Tanto entrada como salida, no se modifican.

La función **porcentaje_xref (&entrada,&salida)** también obtiene el mismo resultado, pero en este caso, en vez de pasar los valores existentes en las variables, pasamos su dirección; por lo que trabajamos directamente con ellas dentro de la función aún cuando las llamemos de diferente manera (**ingresos** y **egresos**). En esta ocasión, las variables globales si se modifican.

Para comprobar esto, después de compilar el programa (que no muestre errores), en vez de ejecutarlo, correr el programa paso a paso.
Colocar un visor de los valores que van tomando las variables. En el entorno Borland C se presiona Ctrl-F7 y se escribe el nombre de la variable que se desea observar. Aparecerá una ventana en la parte inferior del programa en la cual se indicará el estado o valor de la o las variables que se hayan especificado.

Después de haber ejecutado, la función **porcentaje_xvalor**, el contenido de las variables no cambiará mientras que después de haber ejecutado **porcentaje_xref**, salida se verá modificada porque le asignamos un valor dentro de la función.
En realidad, tanto las funciones que utilizan paso de parámetros por valor como las que utilizan el paso por referencia, pueden hacer las mismas cosas. Lo único que cambia es la manera en que trabajan en la memoria de la computadora.
Lo que tenemos que tener en cuenta para saber cuál utilizar, es si queremos que las variables que utilizaremos en el parámetro de la función se modifiquen o no.
No olvidar que las variables que se pasen por lo parámetros sean del mismo tipo de las que están declaradas dentro de los paréntesis de la función.

Creación de Bibliotecas propias

Se pueden utilizar funciones definidas por uno mismo en varios programas, esto se hace creando una biblioteca propia. Esto se logra de la siguiente manera:
-Definir las funciones en un nuevo archivo. Se la llamará librería estándar de C.
-No utilices la función **main().**
-Compilarlo
-Cuando la compilación sea exitosa, se generará un archivo con el mismo nombre que el asignado pero con la terminación **.obj**. Este archivo deberá ser incluido preferentemente en el mismo directorio que se encuentre la biblioteca. En caso contrario, se debe dar la ruta en la sección **Directories** del menú **Options** del entorno Borland C**.**
-En el programa donde se quiere utilizar esta unidad sólo se tendrá que llamarla al principio del programa de la siguiente manera:

#include "nombre_archivo"

Después de esto, se podrá llamar a las funciones que se hayan definido en esa librería sin tener que declararlas al principio del programa.
Además de funciones, en la biblioteca también se pueden definir constantes, macros, etc.
Se plantea al lector pasar a una biblioteca las funciones que para poder utilizarlas posteriormente.

Programa ejemplo:
En el programa que se muestra a continuación, se convierte un número en hexadecimal a su representación decimal. Crear una biblioteca propia para la función hex_dec, compilarlo y ejecutarlo para mejor comprensión.

```
/*Programa que convierte un número en hexadecimal a decimal*/
```

```
float hex_dec(char cadena[])
      {
      short i,j;
      char letra;
      float decimal=0;
      float temp=0;

      i=strlen(cadena);
      for (j=0;i>0;j++,i--)
            {
            letra=cadena[i--];
            switch(letra)
                  {
                  case 1:temp=(1*pow(16,j));break;
                  case 2:temp=(2*pow(16,j));break;
                  case 3:temp=(3*pow(16,j));break;
                  case 4:temp=(4*pow(16,j));break;
                  case 5:temp=(5*pow(16,j));break;
                  case 6:temp=(6*pow(16,j));break;
                  case 7:temp=(7*pow(16,j));break;
                  case 8:temp=(8*pow(16,j));break;
                  case 9:temp=(9*pow(16,j));break;
                  case 0:temp=(0*pow(16,j));break;
                  case 'A':temp=(10*pow(16,j));break;
                  case 'B':temp=(11*pow(16,j));break;
                  case 'C':temp=(12*pow(16,j));break;
                  case 'D':temp=(13*pow(16,j));break;
                  case 'E':temp=(14*pow(16,j));break;
                  case 'F':temp=(15*pow(16,j));break;
                  }
            decimal+=temp;
            }
      return(decimal);
      }


#include<math.h>
#include<string.h>
#include<conio.h>
#include<milibreria.h>


void main(void)
      {
      char hexa[10];
      float numero;

      clrscr();
      printf("Numero hexadecimal (mayúsculas): ");
      gets(hexa);
      numero=hex_dec(hexa);
      printf("\nEn decimal es : %.0f",numero);
      }
```

Ejemplos de aplicación

Variables locales

Ejemplo 1

```
/* Funciones con variables locales */

// En este ejemplo se puede observar el uso de variables locales en cada función
// Variables locales:
//     Función main():     primero, segundo, salida, opcion
//     Función suma():     a,b
//     Función resta():    a,b
//     Función multiplicación():a,b
//     Función división():       a,b

// Todas las funciones excepto main() y division() devuelven valores enteros


#include <stdio.h>


suma(short a, short b);                 /* No especifico tipo  => int */
resta(short a, short b);          /* No especifico tipo  => int */
multiplicacion(short a, short b);       /* No especifico tipo  => int */
float division(short a, short b);       /* Devolverá un número real ya que el tipo es
float */


void main (void)
       {
       short primero, segundo;
       char salida=1;
       char opcion;

       while(salida)
              {
              printf("Ingrese primer operando\n");
              scanf("%d",&primero);
              printf("Ingrese segundo operando\n");
              scanf("%d",&segundo);
              flushall();
              printf("\nSeleccione la función que desea: \n");
              printf("s: SUMA\n");
              printf("r: RESTA\n");
              printf("m: MULTIPLICACION\n");
              printf("d: DIVISION\n");
              printf("S: SALIDA\n");
              scanf("%c",&opcion);
              switch(opcion)
                     {
                     case 's':printf("%d\n",suma(primero,segundo));break;
                     case 'r':printf("%d\n",resta(primero,segundo));break;
                     case 'm':printf("%d\n",multiplicacion(primero,segundo));break;
                     case 'd':printf("%.2f\n",division(primero,segundo));break;
                     case 'S':salida=0;break;
                     default: printf("\nIngrese la opción correcta\n");break;
                     }
              }
       }

suma(short a, short b)
       {
       a+=b;
       return(a);
       }

resta(short a, short b)
       {
       a-=b;
       return(a);
       }

multiplicacion(short a, short b)
```

```
        {
        a*=b;
        return(a);
        }

float division(short a, short b)
        {
        float cociente;
        cociente=(float)a/b;            // Realizo "cast"
                                        // para permitir division real //
        return(cociente);
        }
```

Ejemplo2

```
/* Funciones con variables locales */

// En este ejemplo se puede observar el uso de variables locales en cada función
// Variables locales:
//     Función main():     a, b, salida, opcion
//     Función suma():     a,b
//     Función resta():    a,b
//     Función multiplicación():a,b
//     Función división():       a,b

// Las variables locales tienen como ámbito de validez la función en la que se
definieron,
// por lo que el ejemplo es totalmente válido y funciona sin problemas

// Todas las funciones excepto main() y division() devuelven valores enteros


#include <stdio.h>


suma(short a, short b);          /* No especifico tipo  => int */
resta(short a, short b);         /* No especifico tipo  => int */
multiplicacion(short a, short b);       /* No especifico tipo  => int */
void division(short a, short b); /* El tipo es void ya que no retorna ningún valor */


void main (void)
        {
        char salida=1;
        char opcion;
        short a,b;

        while(salida)
                {
                printf("Ingrese primer operando\n");
                scanf("%d",&a);
                printf("Ingrese segundo operando\n");
                scanf("%d",&b);
                flushall();
                printf("\nSeleccione la función que desea: \n");
                printf("s: SUMA\n");
                printf("r: RESTA\n");
                printf("m: MULTIPLICACION\n");
                printf("d: DIVISION\n");
                printf("S: SALIDA\n");
                scanf("%c",&opcion);
                switch(opcion)
                        {
                        case 's':printf("%d\n",suma(a,b));break;
```

```
                    case 'r':printf("%d\n",resta(a,b));break;
                    case 'm':printf("%d\n",multiplicacion(a,b));break;
                    case 'd':division(a,b);break;
                    case 'S':salida=0;break;
                    default: printf("\nIngrese la opción correcta\n");break;
                    }
                }
        }

suma(short a, short b)
        {
        a+=b;
        return(a);
        }

resta(short a, short b)
        {
        a-=b;
        return(a);
        }

multiplicacion(short a, short b)
        {
        a*=b;
        return(a);
        }

void division(short a, short b)
        {
        float cociente;
        cociente=(float)a/b;            // Realizo "cast"
                                    // para permitir division real //
        printf("%.2f\n",cociente);
        }
```

Variables globales

Ejemplo1

```
/* Funciones con variables globales */

// En este ejemplo se puede observar el uso de las variables globales a y b
// declaradas fuera de cualquiera de las cinco funciones.
// Variables locales:
//     primero, segundo, salida, opcion. Todas de la función main()
// Variables  globales:
//     a,b

// Todas las funciones excepto main() y division() devuelven valores enteros


#include <stdio.h>


suma(void);         /* No especifico tipo  => int */
resta(void);        /* No especifico tipo  => int */
multiplicacion(void);     /* No especifico tipo  => int */
float division(void);     /* El tipo es float ya que devolverá un número real */


short a,b;
```

```
void main (void)
	{
	short primero, segundo;
	char salida=1;
	char opcion;

	while(salida)
		{
		printf("Ingrese primer operando\n");
		scanf("%d",&primero);
		a=primero;
		printf("Ingrese segundo operando\n");
		scanf("%d",&segundo);
		b=segundo;
		flushall();
		printf("\nSeleccione la función que desea: \n");
		printf("s: SUMA\n");
		printf("r: RESTA\n");
		printf("m: MULTIPLICACION\n");
		printf("d: DIVISION\n");
		printf("S: SALIDA\n");
		scanf("%c",&opcion);
		switch(opcion)
			{
			case 's':printf("%d\n",suma());break;
			case 'r':printf("%d\n",resta());break;
			case 'm':printf("%d\n",multiplicacion());break;
			case 'd':division();break;
			case 'S':salida=0;break;
			default: printf("\nIngrese la opción correcta\n");break;
			}
		}
	}

suma(void)
	{
	a+=b;
	return(a);
	}

resta(void)
	{
	a-=b;
	return(a);
	}

multiplicacion(void)
	{
	a*=b;
	return(a);
	}

float division(void)
	{
	float cociente;
	cociente=(float)a/b;            // Realizo "cast"
                              // para permitir division real //
	return(cociente);
	}
```

Ejemplo2

```
/* Funciones con variables locales cuando se pretendían globales */
```

```
// Aquí se verá algo que no se puede hacer ya que dará un error de compilación.
// En el ejemplo anterior a y b son variables globales y en este también se pretende que
a y b
// sean globales pero dado que se definen dentro de la función main(), son locales a ésta
y el
// resto de las funciones no pueden hacer uso de esas variables.


#include <stdio.h>


suma(void);               /* No especifico tipo  => int */
resta(void);              /* No especifico tipo  => int */
multiplicacion(void);     /* No especifico tipo  => int */
void division(void);            /* No retorna ningún valor ya que el tipo es void */


void main (void)
      {
      short primero, segundo;
      char salida=1;
      char opcion;
```

~~short a,b;~~ NO!!!

```
      while(salida)
            {
            printf("Ingrese primer operando\n");
            scanf("%d",&primero);
            a=primero;
            printf("Ingrese segundo operando\n");
            scanf("%d",&segundo);
            b=segundo;
            flushall();
            printf("\nSeleccione la función que desea: \n");
            printf("s: SUMA\n");
            printf("r: RESTA\n");
            printf("m: MULTIPLICACION\n");
            printf("d: DIVISION\n");
            printf("S: SALIDA\n");
            scanf("%c",&opcion);
            switch(opcion)
                  {
                  case 's':printf("%d\n",suma());break;
                  case 'r':printf("%d\n",resta());break;
                  case 'm':printf("%d\n",multiplicacion());break;
                  case 'd':division();break;
                  case 'S':salida=0;break;
                  default: printf("\nIngrese la opción correcta\n");break;
                  }
            }
      }

suma(void)
      {
      a+=b;
      return(a);
      }

resta(void)
      {
      a-=b;
      return(a);
      }

multiplicacion(void)
      {
```

```
        a*=b;
        return(a);
        }

void division(void)
        {
        float cociente;
        cociente=(float)a/b;            /* Realizo "cast" para permitir division real */
        printf("%.2f\n",cociente);
        }
```

Arreglos en C

Arreglos

También conocidos con el nombre original del inglés **ARRAY**,

> **un arreglo es un conjunto de elementos del mismo tipo agrupados en una sola variable.**

Para ingresar a un elemento en particular, utilizamos un índice. Existen arreglos unidimensionales, bidimensionales y tridimensionales.
Su uso más común es en la implementación de cadenas de caracteres. Recuerda que en C no existen variables de tipo cadena por lo cual se utiliza un arreglo de caracteres.
Físicamente, un arreglo es un conjunto de localidades de memoria contiguas donde la dirección más baja corresponde al primer elemento y la dirección más alta al último.
En un arreglo de n elementos, éstos ocuparan desde la casilla 0 hasta la n-1.

I M P O R T A N T E

Por si mismo, el nombre del arreglo apunta a la dirección del primer elemento del arreglo.

Arreglos Unidimensionales (Vectores)

La forma general para definir un arreglo de sólo una dimensión es la siguiente:

```
tipo_dato  nombre_vector
[tamaño]
```

tipo_de_dato se refiere al tipo de dato de cada elemento del arreglo y **tamaño** es la cantidad de elementos agrupados en la misma variable.

Arreglos Multidimensionales (Matrices)

Así como en el álgebra existen vectores y matrices, en C, podemos crear arreglos de dos o más dimensiones. El límite de dimensiones, viene dado por el compilador. Su forma general de declaración es

tipo_dato variable [long1] [long2] ... [longN]

tipo_dato es el tipo de dato de los elementos del arreglo
long1, long2 ... longN es la longitud de cada dimensión del arreglo.

Este tipo de arreglos no se utiliza muy frecuentemente debido al gran espacio en memoria que ocupan. Otra desventaja es que el acceso a un arreglo multidimensional dura más tiempo que el requerido por uno del tipo unidimensional.
Cuando se pasan arreglos multidimensionales a funciones, se tiene que declarar todo excepto la primera dimensión.
Por ejemplo:

```
#include<stdio.h>
funcion1(short multiarreglo[][3][[4][5])
        {
        .
        .
        .
        }
main()
        {
        short m[2][3][4][5];
        funcion(m[][3][4][5]);
        .
        .
        .
        }
```

Claro que si se desea, se puede especificar también la longitud de la primera dimensión.

Forma de acceso a un elemento específico del arreglo

Para acceder a uno de los elementos del arreglo en particular, basta con invocar el nombre del arreglo y especificar entre corchetes el número de casilla que ocupa el elemento en el arreglo.

Por ejemplo, si queremos acceder al cuarto elemento de un arreglo de 10, se invocaría de la siguiente manera:

nombre_variable[3]

I M P O R T A N T E

Recuerdar que el arreglo almacena desde la casilla 0. Por tanto, en un arreglo de 10 casillas, éstas están numeradas del 0 al 9.

Strings – cadenas

El uso más común de los arreglos unidimensionales es la implementación de una cadena (conjunto) de caracteres porque recuerde que en C no existe este tipo de datos.

Utilización de arreglos unidimensionales como cadenas

Una cadena se define en C como un arreglo de caracteres cuyo último elemento es el caracter nulo ('\0')

Por esta razón es necesario que al declarar los arreglos, estos posean un caracter más que la cadena más larga que pueda contener.
Por ejemplo si deseamos crear un cadena que contenga 5 caracteres la declaración debe hacerse como sigue:
char cadena[6];
Esto es con el fin de dejar el último espacio para el caracter nulo.
No es necesario añadir explícitamente el caracter nulo de las constantes de cadena porque el compilador de C lo hace automáticamente.

Inicialización de arreglos con tamaño

En C, podemos inicializar (dar un valor determinado a la variable antes de usarla) arreglos globales y arreglos estáticos locales en el momento de declararlos. No es posible inicializar arreglos globales no estáticos.
Su forma general de inicialización es

tipo_dato variable [tamaño1] [tamaño2]...[tamaño] = {lista de valores};

lista de valores es un conjunto de constantes, separadas comas, cuyo tipo es compatible con **tipo_dato**. La primera constante se coloca en la primera posición del arreglo, la segunda constante en la segunda posición, y así sucesivamente. Fíjese que un punto y coma sigue a **}**.

A continuación tenemos la inicialización de un arreglo unidimensional:

```
short digitos[5]={'0','1','2','3','4','5','6','7','8','9'};
```

En el caso de los arreglos unidimensionales de caracteres podemos inicializarlos abreviadamente con la forma:

```
char variable[tamaño]="cadena";
```

Por ejemplo:

```
char nombre[6]="clase";
```

Lo anterior es lo mismo que si inicializaramos **nombre** caracter a caracter como en el ejemplo de dígitos.

```
char nombre[6]={'c','l','a','s','e'.'\0'};
```

I M P O R T A N T E :

Se debe estar seguro de que el arreglo que se declara es suficientemente largo para incluirlo. Esto es por lo que **nombre** tiene 6 caracteres de longitud en vez de 5 que es la cantidad de letras ya que cuando se utiliza una cadena constante, el compilador proporciona la terminación nula automáticamente.

Inicialización de arreglos sin tamaño

En los arreglos con tamaño, tenemos que calcular que la longitud del arreglo sea lo suficientemente grande para que puedan ser almacenados todos los elementos que deseábamos. Si tuviéramos que inicializar varios arreglos de cadena sería fastidioso contar cuantos caracteres ocupa cada arreglo. Es posible que C calcule automáticamente la longitud del arreglo utilizando **la inicialización de arreglos indeterminados** la cual permite que el compilador de C cree automáticamente un arreglo suficientemente grande para mantener todos los inicializadores presentes si el tamaño del arreglo no está especificado.
El uso de la inicialización de los arreglos indeterminados permite al programador cambiar el contenido de cualquiera de las cadenas sin tener que reconsiderar el tamaño del arreglo. También puede utilizarse también en arreglos multidimensionales (2 o más). En este caso, se debe especificar todo, sin considerar la dimensión que se encuentra más a la izquierda para permitir al compilador de C indexar el arreglo adecuadamente. El método es similar a la especificación de parámetros de un arreglo. De este modo se pueden construir tablas de longitudes variables, y el compilador asignará automáticamente el espacio suficiente para ellas.
La ventaja de este tipo de declaración sobre la versión del tamaño determinado es que la tabla puede alargarse o acortarse sin cambiar las dimensiones del arreglo.

Funciones para manejo de cadenas

gets()

Esta función constituye por su simpicidad, una rápida manera de ingreso de caracteres alfanuméricos a una cadena (string) o lo que es lo mismo, a un vector de caracteres.

```
#include<stdio.h>
char gets(char cad[])
```

Descripción
Esta función lee una cadena de caracteres desde el teclado y la coloca en la cadena **cad**. Se leen caracteres hasta que se recibe la pulsación de <ENTER>. Esto no quiere decir que en la cadena se almacenará este caracter sino que añadirá un caracter nulo (\0) para identificar el final de la cadena. En caso de error, **gets()** retorna un puntero nulo (NULL) y el contenido de cad será indeterminado.Con **gets()** se pueden leer todos los caracteres que se deseen. Por lo tanto, corresponde al programador asegurarse que la cantidad de caracteres leídos no superen la capacidad del array.
Ahora veamos como tipear la entrada con gets:

```
.
.
char apellido[31];
gets (apellido);    /*Ingresa apellido por teclado*/
.
.
```

puts()

```
#include<stdio.h>
int puts(char cad[])
```

Descripción
La función **puts()** escribe en pantalla el contenido de una cadena apuntada por **cad** (un conjunto de caracteres) y posiciona el cursor en la siguiente línea.
Cómo tipear la salida con puts:

```
.
.
char apellido[31];
gets (apellido);    /*Ingresa apellido por teclado*/
puts (apellido);    /*Muestra apellido en pantalla*/
.
.
```

scanf() y especificador %s

Para leer la cadena de caracteres y almacenarla directamente en el array se utiliza la función scanf y formateando la entrada con %s. De esta manera la función salta los espacios en blanco (hablamos de espacios y tabuladores) y guarda en el array de caracteres todo lo ingresado alfanuméricamente hasta que como último caracter se ingrese un enter. **scanf()** añadirá automáticamente un caracter nulo al final de la cadena (‘\0’).
Es importante destacar que el tamaño del array debe ser definido teniendo en cuenta la cadena que será ingresada. Por ejemplo, para un vector que guarde apellidos sería necesario reservar 30

elementos.
Ahora veamos como tipear la entrada con scanf:

```
.
.
char apellido[30];
scanf (“%s”, apellido);
.
.
```

I M P O R T A N T E

El vector apellidos no está precedido por & dentro de scanf. Esto se debe a que scanf es una función que se ha escrito esperando recibir la dirección de una variable. El operador & puesto por delante de la variable devuelve la dirección inicio que dicha variable ocupa en memoria. A partir de esa dirección inicio se reservan bytes de memoria de acuerdo al tipo definido (recordar por ej. 2 bytes para los ‘short’). Es por esto que se precede el signo & en un ingreso a variables. En el caso de los vectores, el nombre del vector ya es en si mismo la dirección del primer elemento, por lo que no hace falta que sea precedido por & dentro del scanf.

---

printf() y especificador %s

La función printf con ‘%s’ espera que el argumento sea una cadena de caracteres, y la misma se imprime excluyendo al caracter nulo característico de fin de string (cadena).
Cómo se utiliza:

```
.
.
char apellido[30];
scanf (“%s”, apellido);
printf (“%s”, apellido);
.
.
```

strcmp(), strcpy(), strcat(), strlen()

La librería string.h proporciona varias funciones para trabajar con cadenas. Algunas de las principales funciones que soporta C son las siguientes:

Función	Descripción
strcpy (S1,S2)	**Copia S2 en S1**
strcat (S1,S2)	**Concatena S2 al final de S1**
strlen (S1)	**Devuelve la longitud de S1**

strcmp (S1,S2)	**Compara S1 y S2. Según el caso, devolverá:** **S1 = S2 ⇒ 0** **S1 < S2 ⇒ negativo** **S1 > S2 ⇒ positivo**

Funciones que utilizan vectores globales

La forma de pasar un arreglo a una función consiste en llamar a la función y en el argumento, especificar el nombre del arreglo sin ninguna indexación. Esto hace que se pase a la función la dirección del primer elemento del arreglo ya que en C no es posible pasar el arreglo completo como argumento.
Por ejemplo:

```
main()
       {
       short conjunto[20];
       clrscr();
       .
       .
       .
       funcion(conjunto);
       .
       .
       .
       }
```

Aquí, al pasar el arreglo **conjunto** a **función**, estamos pasando la dirección en memoria del primer elemento de **conjunto**. En caso de que dentro de la función tuviésemos que acceder a algún elemento del arreglo, se pasa de la misma manera sólo que dentro de la función utilizaremos los corchetes para acceder al elemento deseado.
Hay tres formas de declarar un arreglo como parámetro formal: como un arreglo delimitado, como un arreglo no delimitado y como un puntero.
Por ejemplo:

```
#include<stdio.h>
funcion1(short conjunto[20]) /*Delimitando el array*/
       {
       .
       .
       .
       }
o como
funcion1(short conjunto[]) /*arreglo no delimitado*/
       {
       .
       .
       .
       }
```

o se puede declara como

```
funcion1(short *conjunto) /*como un puntero*/
       {
       .
       .
       .
       }
```

El resultado de los tres métodos de declaración es idéntico.

Nota complementaria: Las funciones gets y puts

Con el objetivo de completar el tema, se dá a título de ejemplo una implementación de las funciones gets y puts. Este código fué compilado en Code::Blocks 13.12-RC2.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

void migets(unsigned char *texto, char tamano);
void miputs(unsigned char *texto);

unsigned char cadena[30];

int main()
{
   migets(cadena,30);
   miputs(cadena);

   return 0;
}

void migets(unsigned char *texto,char tamano)
{
   unsigned char i=0;
   char letra;

   letra=getch();
   while(letra!=13 && i<tamano-1) /* 13 es la tecla ENTER */
   {
      texto[i]=letra;
      i++;
      letra=getch();
   }
   texto[i]=NULL;
}

void miputs(unsigned char *texto)
{
   unsigned char i=0;
   while(texto[i]!=NULL)
   {
     printf("%c",texto[i]);
     i++;
   }

}
```

Algoritmos de ordenamiento de vectores

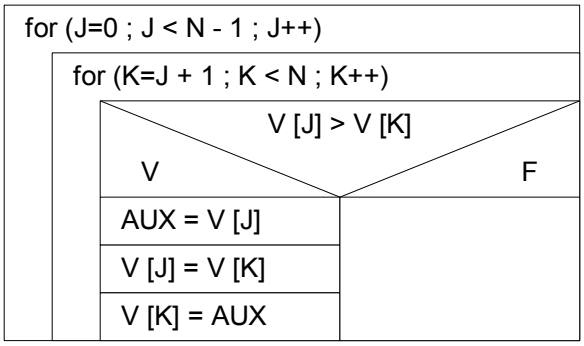
Ordenar es la operación de arreglar los registros de un vector con algún orden secuencial de acuerdo a un criterio de ordenamiento.
Podemos hablar de ordenamiento en forma ascendente (el primer elemento es el más chico, y el último el más grande) y en forma descendente (el primer elemento será el mayor de todos, y el último el menor).
Todo ordenamiento se efectúa con base en el valor de algún campo en un registro. Por ejemplo podemos citar las guías telefónicas, donde están los datos están ordenados alfabéticamente por apellido del cliente.
El propósito principal de un ordenamiento o (del inglés) sort es el de facilitar las búsquedas de los miembros del conjunto ordenado.¿Cuándo conviene usar un sort? Cuando se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo.

Distintos métodos de ordenamiento que se ejecutan en RAM o internos:

- Burbuja (Pivote, Simple)
- Burbuja (mejorado)
- Selection Sort
- Heap Sort
- Insert Sort
- Merge Sort (Breadth First, Input List size is a power of 2)
- Merge Sort (Depth First, Input List size is not a power of 2)
- Quick Sort

Método del pivote

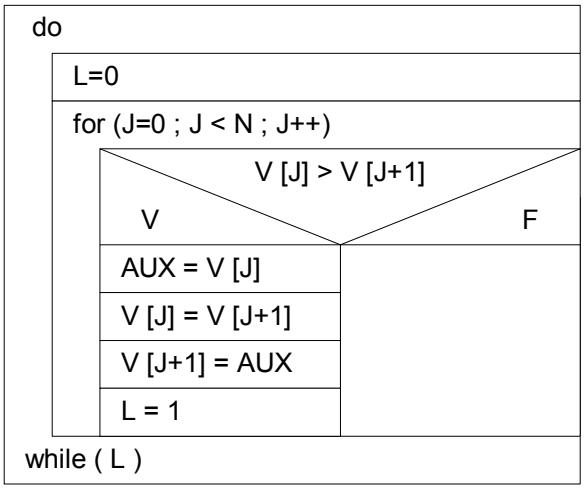
Es el método más simple. Se basa en tomar un elemento (que considera como pivote) e ir comparándolo con el resto de los elementos.
Trabaja de la siguiente manera:
El primer elemento es comparado con el segundo, luego con el tercero y así sucesivamente con los subsecuentes elementos. Si cualquiera de esos otros elementos resulta ser menor que el elemento que se toma como **pivote** o patrón para la comparación, entonces se intercambia la ubicación de los elementos dentro del vector. Eventualmente, luego de ser considerado el último elemento del array, e intercambiado si fuera necesario, podemos afirmar que el primer elemento del vector es el elemento más pequeño.
El paso sguiente es entonces, repetir el mismo proceso pero tomando al segundo elemento como pivote. Luego será considerado como pivote al tercero, y así sucesivamente. Así, cuando el ante-último elemento fue el pivote, el ordenamiento concluye.
El diagrama Chapin correspondiente es:



Este algoritmo de ordenamiento es del tipo progresivo, ya que en forma ascendente va moviendo los elementos del vector y los deja ordenados. La progresión va desde el comienzo hacia el final del vector.

Método de la burbuja

Ahora analicemos el siguiente diagrama que ordena un vector V de N elementos:

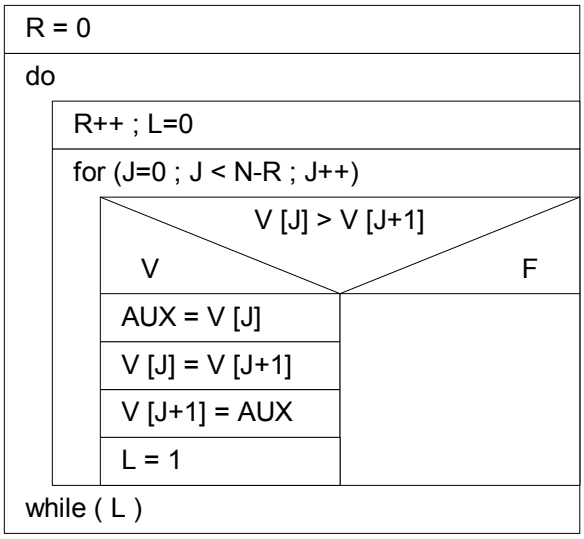


Está implementado en base a un ciclo do-while que ejecuta en su interior un lazo for.
El método de burbuja va comparando cada elemento del arreglo con el siguiente (ver la comparación entre V[J] y V[J+1]); si un elemento es mayor que el que le sigue, entonces se intercambian; ésto producirá que en el arreglo quede como su último elemento, el más grande. Este proceso deberá repetirse recorriendo todo el arreglo hasta que no ocurra ningún intercambio. Los elementos que van quedando ordenados ya no se comparan. El mayor, en cada pasada va a la última posición, de allí el nombre de burbuja que se le dio al método.

Método de la burbuja mejorado

Veamos una mejora al método de la burbuja. Se ha introducido en el diagrama a la variable **R**. Ésta, antes de emprenderse el ordenamiento propiamente dicho (lazo do-while) debe ser inicializada en 0, ya que su misión es la de optimizar el algoritmo aprovechando una característica del método de la

burbuja: ***al terminar el ciclo for siempre queda el mayor en la última posición del vector***.
De esta manera, si cada vez que ejecutamos el for, vamos decrementando la cantidad de veces que itera en una unidad con respecto a la pasada anterior, hemos optimizado el algoritmo. Este es el motivo por el cual R se incrementa fuera del lazo for, y aparece disminuyendo a N.
El diagrama de Chapin es:



Comparación de los métodos (Eficiencia en tiempo de ejecución)

Una medida de la eficiencia es

- Contar el número de comparaciones (C)
- Contar el número de movimientos de items (M)

Estos están en función del número de items a ser ordenados.
Un "buen algoritmo" de ordenamiento requiere de un orden n *log n* comparaciones.

Método del pivote

El total de comparaciones que realiza es

(N - 1) + (N - 2) + (N - 3) + . . . + N – (N - 1)

o también

$$\sum_{i=1}^{N-1}(N-i)=\frac{N^2-N}{2}$$

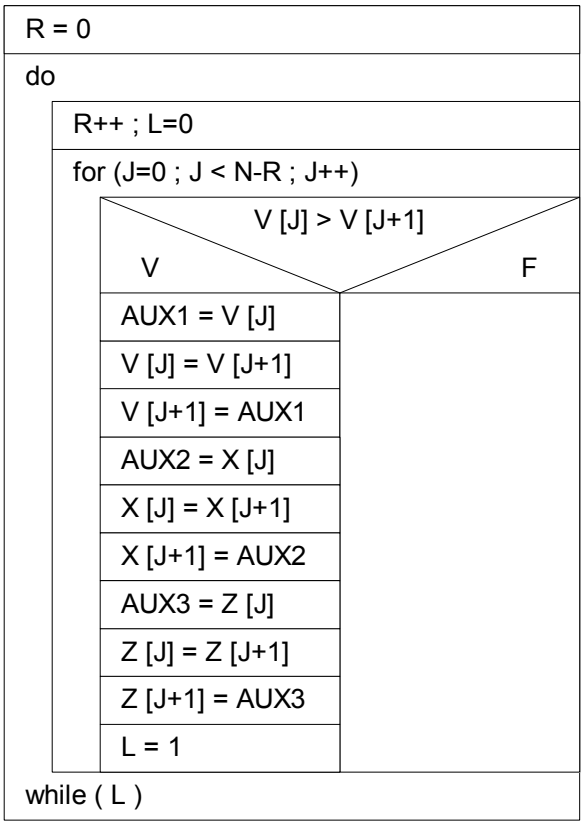
Método burbuja mejorado

El total de comparaciones variará según el grado de desorden.
El peor caso está dado por si el menor está en la última posición

$$(N-1) \leq Totaldecomparaciones \leq \frac{N^2 - N}{2}$$

Arrastre de vectores

Muchas veces tenemos varios vectores que contienen información que se relaciona a través del los subíndices de los mismos. Por ejemplo, en un vector V podemos tener los legajos de los 500 miembros de un club, en otro vector X (también de 500 elementos) disponemos de sus edades y en otro vector Y, los documentos de los socios. Si deseamos ordenar la información (esto ya es una base de datos) deberemos ordenar de acuerdo a la variable que necesitamos ordenada (por ejemplo, el legajo), pero se debe hacer un "arrastre" de los datos de los otros vectores, a fin de que no se desorganicen los datos (es decir que luego del ordenamiento no se hayan cambiado los documentos de las personas).
Veamos cómo es esto:



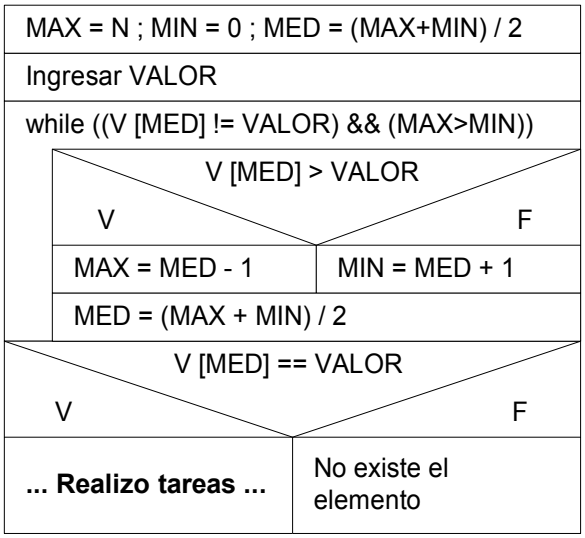
N O T A

Observar que las variables auxiliares se han declarado con distintos nombres, pero en caso de tratarse de datos que sean del mismo tipo se podría utilizar una sola variable, por ejemplo AUX.

Búsqueda binaria

Este diagrama debe adaptarse al programa en el cual se inserte teniendo en cuenta cuál es el vector (V) con el que se trabaja y la cantidad de elementos del mismo, ya que se debe inicializar a la variable MAX.

Observar que la forma de búsqueda es partiendo la cantidad de elementos del vector en dos, y de alli se compara. Si el valor buscado es mayor que el elemento encontrado en la mitad del vector, se buscará en la última mitad de elementos. Luego se irá acotando cada vez más la búsqueda redefiniendo las cotas. De forma análoga ocurre si el valor buscado estuviera en la primera mitad de elementos.
Dado un vector de tamaño T su menor subíndice será 0 (MIN) y el mayor será N=T-1(MAX)

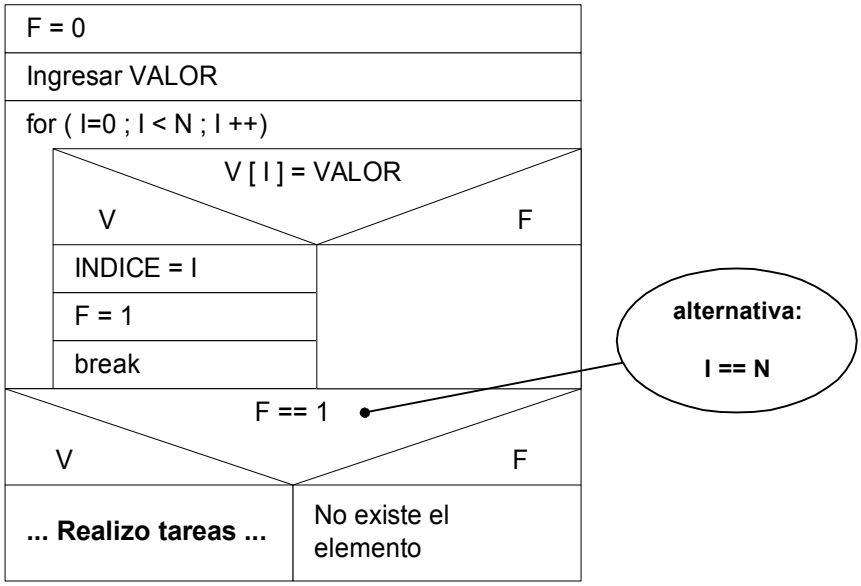


I M P O R T A N T E

Para el algoritmo mostrado es necesario previamente realizar un ordenamiento el vector.

Búsqueda secuencial

En este algoritmo se recorre en forma secuencial todos los elementos del vector y cuando se encuentra el valor se corta el lazo mediante el break. En la variable INDICE quedaría la posición del elemento. Para saber si se encuentra el elemento, se pregunta por la variable F que debería estar seteada en 1 en caso de hallarse dicho elemento.



N O T A

Observar que debido a que el vector se recorre totalmente no es necesario que el mismo haya sido previamente ordenado.

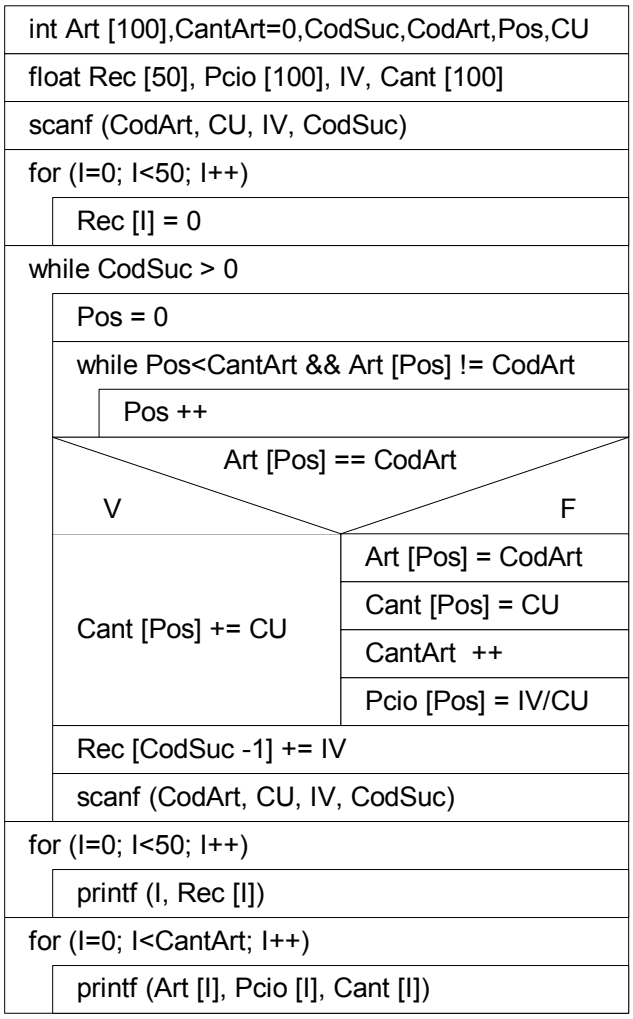
Algoritmos de búsqueda e inserción en vectores

El concepto de estos algoritmos es, dado un determinado vector permitir ingresar valores nuevos (distintos) al vector. Es decir que si el dato que se debe ingresar ya se encuentra en el vector no se lo ingresará, mientras que si el dato no forma parte de los elementos del arreglo, deberá ser ingresado. Es por esto se compone de dos fases, primero la de **búsqueda** (por ejemplo la secuencial ya vista) para ver si el dato ya existe en el vector, en cuyo caso se determinará si debe ser ingresado – **inserción** – o no.

Ejemplo de aplicación

Una empresa comercializa 100 productos. Recibe el resultado de las ventas efectuadas por 50 sucursales, un registro de cada venta con la siguiente estructura:
Código del artículo
Cantidad de unidades vendidas
Código de la sucursal
Importe de la venta
El informe está desordenado y desconoce la cantidad. Se finaliza el ingreso de datos con Código de Sucursal nulo (0).
El programa mostrará la siguiente información:
Los importes recaudados por cada sucursal
Los códigos de artículos con su precio unitario y cantidad de artículos vendidos.
Los códigos de sucursales se numeran de 1 a 50.

A continuación el diagrama correspondiente:



Funciones que convierten cadenas de caracteres a formato numérico

En algunas ocasiones necesitaremos ingresar un dato en binario y convertirlo a un unsigned long (ó long). Hay una función que, ingresando el dato numérico por medio de un string, permite convertir el dato a cualquier base entre 2 y 36. Esta función es strtoul de la stdlib.h. Esta funcion convierte una cadena de caracteres a un unsigned long.
En el siguiente ejemplo convertiremos un binario de hasta 16 bits a su equivalente unsigned long. Si el valor 2 que aparece en la función fuese 10, estaríamos indicando que el número ingresado es un decimal, y si pusiéramos 16, que es un hexadecimal. El valor NULL va por decreto (no lo explicaremos).

```
#include<string.h>
#include<stdlib.h>
#include<conio.h>
#include<stdio.h>

void main(void)
{
        char cadena[17];
        unsigned long numero;
```

```
clrscr();
        gets(cadena);
        numero=strtoul(cadena, NULL, 2);
        printf("%ld",numero);
}
```

Si ingreso la cadena “1101” obtengo 13 y si ingreso 16 unos obtengo 65535.

El siguiente ejemplo “flexibiliza” el ingreso de los números a cualquier base entre 2 y 32. Las letras que forman los números hexadecimales pueden ingresarse tanto en mayúscula como en minúscula.

```
#include<string.h>
#include<stdlib.h>
#include<conio.h>
#include<stdio.h>

void main(void)
{
        char cadena[17];
        unsigned long numero;
        int base;
        clrscr();
        printf("Ingrese el numero a convertir: ");
        gets(cadena);
        printf("La base del numero a convertir es: ");
        scanf("%d",&base);
        numero=strtoul(cadena, NULL, base);
        printf("%ld", numero);
        getch();
}
```

Esta función como vimos convierte una cadena a un unsigned long. Existe otra función idéntica en uso que es strtol que convierte una cadena a un long.

Otras funciones que convierten cadenas de caracteres a valores numéricos son: atoi, atol, atof y strtod.

Importante:

Si recibiéramos los datos desde un puerto, ASCII a ASCII, esto es, en una cadena de caracteres (string), por medio de estas funciones convertiríamos la información recibida a su “equivalente numérico”. Hay que tener en cuenta que tengo que conocer “qué era el original” (si era float, long int, double, etc) para poder “reconstruirlo” al dato correctamente. Además, hay que tener muy en cuenta, que la cadena ASCII recibida pudo haber sido originalmente una cadena ASCII y que no hay que modificarla ni convertirla a nada.

Funciones que convierten números a cadena de caracteres

Estas funciones pueden ser de gran utilidad a la hora de tener que enviar información a un port, ASCII a ASCII (el complemento de lo visto anteriormente).

Función itoa: Permite convertir un entero signado a su equivalente cadena ASCII. Si por ejemplo el número fuese –12345, lo que se almacenará en un string serán los caracteres ASCII:
'-''1''2''3''4''5', además del NULL.

si tenemos

```
int numero;
char vector[15];

itoa(numero, vector, 10);
```

El 10 en el ejemplo indica que el valor a convertir quedará en el string en base 10. La función itoa coloca el NULL en el string, en forma equivalente a lo que hacia la funcion gets. Es además, una forma sencilla de poder convertir un decimal, octal o binario a cualquier base.

Si por ejemplo en numero tengo 16 y el 10 lo reemplazo por 2 en la funcion, lo que me va a quedar en le string es '1''0''0''0''0'.

Hay que observar que el vector generado es mucho mayor que los caracteres del número a almacenar y que una gran parte del mismo quedará con basura que será ingnorada en el momento de la impresión ya que solo se visualizará el contenido hasta encontrar el NULL.

Vamos a ver el siguiente programa:

Supongamos que queremos generar un programa que nos permita convertir un numero N a todas las bases entre 2(binaria) y base 24 (¡sí base 24!!). El programa hecho con itoa será el siguiente:

```
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
  int N = 32767;
  char string[25];
  int base;
  clrscr();
  for(base=2;base<25;base++)
  {
                itoa(N, string, base);
                printf("El %d en base %d queda %s\n", N, base, string);
  }
}
```

No tiene sentido hacer por este metodo la conversión a bases mayores ya que ¡nos quedamos sin letras!!!! (recordar como se forman los números, y en particular los hexadecimales con las letras que van desde la A a la F). Queda para el lector el ver las funciones ltoa y ultoa (la primera para long int y la segunda para unsigned long int).

Capítulo 8

Estructuras de datos
Uniones
Campos de bits

Estructuras de datos (struct)

La estructura es un tipo de datos compuesto. Se define como un conjunto de variables - que pueden ser de distintos tipos de datos -, que se referencian bajo un mismo nombre.
Así, en la estructura, todos estos tipos de datos juntos los tratamos como a una unidad.

Supongamos el caso de necesitar almacenar información de componentes electrónicos y de ellos sabemos el tipo de componente (resistor, capacitor, transistor, etc.), código del mismo, valor, reemplazo, potencia, etc. En este caso precisamos **definir un nuevo tipo de dato** que reúna a todos los datos que poseemos. Entonces haremos una estructura, y le colocaremos un nombre, por ejemplo "componentes".

Hemos formado un nuevo tipo de dato que es precisamente la estructura *componentes* y que podrá reunir o aunar a los datos que hayamos incorporado en la misma.

Veamos cómo declararíamos en C a la estructura mencionada:

```
struct componentes
       {
       char clase[11];
       char codigo[6];
       char reemplazo[11];
       short valor;
       short potencia;
       };

struct componentes componentes_electronicos;
```

struct componentes es el tipo de dato; **clase, codigo**, **reemplazo**, **valor** y **potencia** son los ***campos*** o ***miembros*** de la estructura y **componentes_electronicos** es la variable declarada de ese tipo.

De esto queda en claro que una estructura es un tipo de dato definido por el programador.

Su forma general de definición es

```
struct  nombre_estructura
        {
        tipo nombre_variable;
        tipo nombre_variable;
        .
        }
struct  nombre_estructura
variables_tipo_nombre_estructura;
```

Otra forma usual es la siguiente:

```
struct componentes
       {
       char clase[11];
       char codigo[6];
       char reemplazo[11];
       short valor;
       short potencia;
       } componentes_electronicos;
```

Pero en este caso no se podría tener el tipo global y las variables locales ya que ambas o serían globales o locales.

Veamos otro ejemplo:
Es el caso en el que queremos almacenar la información correspondiente a ciertos archivos de nuestra computadora. De cada archivo registraremos:

- Nombre
- Extensión
- Fecha creación
- Tamaño
- Atributo de archivo sólo lectura
- Atributo de archivo oculto
- Atributo de archivo del sistema

Entonces la declaración en C será:

```
struct info_archivo
       {
       char nombre[9];
       char extension[4];
       char fecha[9];
       short tamanio;
       char archivo_solo_lectura;
       char archivo_oculto;
       char archivo_sistema;
       };
```

Como dijimos, la estructura define un nuevo tipo de dato. Ésta le dará el "tipo" a una variable, en este caso la estructura es *info_archivo* y la variable del tipo info_archivo es *archivo*.
Los ámbitos de validez también se aplican a las estructuras. Entonces podremos tener una estructura definida en forma **global** o en forma **local.**

En la forma global se la define fuera de cualquier función del programa, incluso de la función main(), es decir al comienzo del programa, en donde la validez de la estructura será para todas las funciones del programa, de tal forma que cualquier función tiene acceso a los datos suministrados por la misma.

En la forma local se la define dentro de alguna función del programa, en donde la validez de la estructura será sólo para esa función. Cabe aclarar que si se define a la estructura dentro de la función main(), la misma tendrá un ámbito de validez local a dicha función.

Miembros de la estructura. Operador punto (.)

Los **miembros** o **campos** de la estructura se referencian con la ayuda del operador punto **"."**.
La utilización de este operador se lleva a cabo haciendo mención del nombre de la estructura seguido por un punto y a continuación el campo que se desee. Por ejemplo:

```
archivo.tamanio=900;
```

Aquí se está asignando 900 al campo *tamanio* de la variable archivo.

La forma general de citarlos es:

variable_estructura . campo_elemento

El nombre de la variable tipo estructura seguido del punto y luego del nombre del campo referencia ese campo individual de la estructura.
Así, para imprimir en pantalla el elemento *tamanio* de la variable archivo, escribimos:

```
printf("%d",archivo.tamanio);
```

De la misma forma podemos utilizar gets() para obtener un valor para *archivo.extensión* como se muestra aquí:

```
gets(archivo.extension);
```

I M P O R T A N T E :

Para el ingreso de cadenas de caracteres podemos utilizar ya sea la función *scanf()* o *gets()*, ambas de la librería *stdio.h*. Cabe mencionar que presentado el caso de tener que hacer varios ingresos será necesario llamar a la función ***fflush(stdin)*** antes de cada ingreso que coloquemos.

fflush(stream) es también una función de la librería *stdio.h* y se utiliza para limpiar corrientes de información (stream) las cuales son buffers en la memoria de lectura escritura donde en el caso de un ingreso de una cadena de caracteres la misma es la corriente estándar de entrada (stdin).
Vale recordar que la función gets() acepta el espacio como un caracter válido en cambio utilizando scanf() el espacio se interpreta como fin de cadena.

Ahora bien, volviendo al ejemplo mostrado, se ha definido la estructura y la variable asociada. Veamos otra forma de declararla:

```
#include<stdio.h>

struct info_archivo
      {
      char nombre[9];
      char extension[4];
      char fecha[9];
      short tamanio;
      char archivo_solo_lectura;
      char archivo_oculto;
      char archivo_sistema;
      };

void main(void)
       {
       struct info_archivo archivo;
       }
```

En este caso se ha definido la estructura en forma global y luego en la función main se define a *archivo* como *info_archivo*. Entonces, *archivo* es una variable local de la función main, mientras que el tipo de estructura es global.

Estructuras anidadas

Para organizar mejor los datos de nuestro ejemplo, separaremos los datos de la fecha del archivo, para lo cual haremos la definición de una nueva estructura a la que llamaremos “info_fecha”. Luego esta estructura formará parte de la estructura general “info_archivo”:

```
#include<stdio.h>

struct info_fecha
       {
       short dia;
       short mes;
       short anio;
       };

struct info_archivo
       {
       char nombre[9];
       char extension[4];
       struct info_fecha fecha;
       short tamanio;
       short archivo_solo_lectura;
       short archivo_oculto;
       short archivo_sistema;
       };
```

```
struct info_archivo archivo;
```

Aquí hemos ***anidado*** estructuras. Ahora veamos cómo referenciar a los campos de la estructura anidada:

```
printf("%d", archivo.fecha.anio);
```

O para el ingreso de datos:

```
scanf("%d", &archivo.fecha.anio);
```

El ejemplo completo sería:

```
#include<stdio.h>

struct info_fecha
       {
       short dia;
       short mes;
       short anio;
       };

struct info_archivo
       {
       char nombre[9];
       char extension[4];
       struct info_fecha fecha;
       short tamanio;
       short archivo_solo_lectura;
       short archivo_oculto;
       short archivo_sistema;
       };

struct info_archivo archivo;

void main(void)
       {
       clrscr();
       printf("Ingrese dia ");
       scanf("%d",&archivo.fecha.dia);
       printf("Ingrese mes ");
       scanf("%d",&archivo.fecha.mes);
       printf("Ingrese año ");
       scanf("%d",&archivo.fecha.anio);
       printf("\nFecha ingresada: %02d/%02d/%04d", archivo.fecha.dia, archivo.fecha.mes,
archivo.fecha.anio);
       }
```

Observar que se ha formateado la salida de printf() para una mejor visualización cuando se ejecuta el programa:

%02d: escribe dos cifras en pantalla (para números con menos de 2 cifras completa con ceros a la izquierda)

%04d: escribe cuatro cifras en pantalla (para números con menos de 4 cifras completa con ceros a la izquierda)

Vectores de estructuras

En este momento llegamos a la más importante utilidad de las estructuras: los arreglos de estructuras.
Un vector (array) de estructuras se utiliza con la finalidad de poder manejar diversos tipos de datos y en tal caso poder realizar una sencilla base de datos.
Por ejemplo si llevamos una base con los datos de los empleados de una empresa tal como D.N.I., legajo, sueldo, descuentos jubilatorios, etc., podremos implementarlo con varios vectores que deberán mantener la relación entre ellos. Cada vector podría tener un determinado tipo de dato y así poder registrar la información.
Es más, también podría implementarse con vectores multidimensionales (si los datos fueran del mismo tipo).
Ahora bien, se plantea una nueva necesidad: ¿cómo hacer para ingresarle a esta base de datos los nombres, apellidos, dirección, etc. de los empleados?. Es decir, ¿cómo se podrán almacenar cadenas de caracteres en nuestra base? Sabemos bien que las cadenas de caracteres son en sí vectores, entonces es aquí donde aparece la necesidad de "estructurar" la información ya que las estructuras nos permitirán llevar a cabo este nuevo planteamiento.
Es importante destacar que la ocupación de memoria de un vector de estructuras versus el uso de vectores independientes es exactamente la misma con la gran ventaja de que como una estructura es una variable se simplifica la tarea cuando ordenamos información ya que se intercambian estructuras completas evitándose la tediosa labor de arrastrar todos los elementos en forma individual.

Volvamos al ejemplo de los archivos de la computadora:

```
#include<stdio.h>

struct info_fecha
       {
       short dia;
       short mes;
       short anio;
       };

struct info_archivo
       {
       char nombre[9];
       char extension[4];
       struct info_fecha fecha;
       short tamanio;
       short archivo_solo_lectura;
       short archivo_oculto;
       short archivo_sistema;
       };

struct info_archivo ARCHIVOS[10];
```

Vemos que se ha definido un vector de 10 elementos. Cada elemento del vector es una estructura donde se almacenarán todos los datos que precisábamos.

Veámos un ejemplo completo:

```
#include<stdio.h>
#define cantidad 100

struct info_fecha
       {
     short dia;
       short mes;
```

```
       short anio;
       };

struct info_archivo
       {
       char nombre[9];
       char extension[4];
       struct info_fecha fecha;
       short tamanio;
       short archivo_solo_lectura;
       short archivo_oculto;
       short archivo_sistema;
       };

struct info_archivo ARCHIVOS[cantidad];

void main(void)
{
short indice;
clrscr();

/* Lazo para la entrada de datos */

for(indice=0;indice<cantidad;indice++)
       {
       printf("Ingrese nombre ");
       fflush(stdin);
       gets(ARCHIVOS[indice].nombre);
       printf("Ingrese extension ");
       fflush(stdin);
       gets(ARCHIVOS[indice].extension);
       printf("Ingrese tamaño ");
       fflush(stdin);
       scanf("%d",&ARCHIVOS[indice].tamanio);
       printf("Ingrese día de creación ");
       scanf("%d",&ARCHIVOS[indice].fecha.dia);
       printf("Ingrese mes de creación ");
       scanf("%d",&ARCHIVOS[indice].fecha.mes);
       printf("Ingrese año de creación ");
       scanf("%d",&ARCHIVOS[indice].fecha.anio);
       printf("Atributo solo_lectura? (0/1) ");
       scanf("%d",&ARCHIVOS[indice].archivo_solo_lectura);
       printf("Atributo oculto? (0/1) ");
       scanf("%d",&ARCHIVOS[indice].archivo_oculto);
       printf("Atributo sistema? (0/1) ");
       scanf("%d",&ARCHIVOS[indice].archivo_sistema);
       }

/* Lazo para la presentación de datos */

for(indice=0;indice<cantidad;indice++)
       {
      printf("\nArchivo: %8s.%3s\tTamaño: %d\tFecha: %02d/%02d/%04d",
      ARCHIVOS[indice].nombre, ARCHIVOS[indice].extension,
      ARCHIVOS[indice].tamanio,ARCHIVOS[indice].fecha.dia,
      ARCHIVOS[indice].fecha.mes,ARCHIVOS[indice].fecha.anio);

      printf("\nAtributos: Sololectura:\t %d", ARCHIVOS[indice].archivo_solo_lectura);

      printf("\n           Oculto:\t %d", ARCHIVOS[indice].archivo_oculto);
      printf("\n           Sistema:\t %d", ARCHIVOS[indice].archivo_sistema);
       }
}
```

Ordenamiento y estructuras

Es común la necesidad de trabajar con listados en donde figuran códigos de artículos numéricos, códigos alfanuméricos (aquellos que se conforman de letras y números), números de documentos, cantidades, nombres de personas, etc.
Estamos bien familiarizados con el tratamiento de todo lo que sean cantidades (números), es por esto que dedicamos este punto especialmente para el ingreso, manipulación, ordenamiento y salida por pantalla de caracteres alfanuméricos.
Como hemos visto, en todo ordenamiento hay dos componentes típicos:

- Comparación
- Intercambio de elementos

En el caso de la comparación entre los elementos a ordenar, las variables numéricas directamente utilizan los operadores relacionales '<', '==', '>', pero cuando se trata de letras o letras y números combinados, recurriremos a **strcmp() -** función **string compare -** .

Cuando hay que intercambiar elementos hacemos un intercambio ayudándonos de una variable auxiliar que es del mismo tipo que el vector. Para las variables numéricas, el operador de asignación es lo correcto ('='), y en las variables alfanuméricas **strcpy()** - función **string copy** - nos realizará la copia en la variable auxiliar.

S I M P L I F I C A C I O N

Se aconseja el uso de estructuras ya que brindan simplicidad en el intercambio de elementos, a la vez que se organiza mejor la estructura de nuestro algoritmo. Es decir, veremos que con estructuras no hace falta el uso de strcpy para variables alfanuméricas; simplemente se asignan tal cual lo hacemos con variables numéricas.

Ejemplo de aplicación

Se ingresan por teclado treinta edades y los correspondientes nombres utilizando estructuras. Emitir un listado ordenado alfabéticamente en donde se muestren las edades de los alumnos.

```
# include <stdio.h>
# include <string.h>
# include <conio.h>

# define max 30
struct z
       {
       short edad;
       char nombre[20];
       };

struct z alum[max],aux;
short i,k=0,j;

void main(void)
       {
       clrscr();
       printf("ingrese nombre: ");
       gets(alum[k].nombre);
```

```
	while (strcmp(alum[k].nombre, "salir")!= 0)
		{
		printf("edad:");
		scanf("%d",&alum[k].edad);
		k++;
		printf("ingrese nombre: ");
		fflush(stdin);
		gets(alum[k].nombre);
		}

	for (i=0;i<k-1;i++)
		for (j=i+1;j<k;j++)
			if (strcmp(alum[i].nombre,alum[j].nombre)>0)
				{
				aux=alum[i];
				alum[i]=alum[j];
				alum[j]=aux;
				}

	printf("\nOrdenado por nombres\n");
	for (i=0;i<k;i++)
		printf("\nNombre: %20s Edad: %2d",alum[i].nombre, alum[i].edad);
	}
```

I M P O R T A N T E

Se debió hacer uso de **fflush()** (perteneciente a la librería stdio.h) luego de **scanf()** para limpiar buffers y poder usar **gets().**

Ahora, se ha resuelto el programa de igual forma, pero el ingreso de los strings es a través de **scanf().** Observar que no fue necesaria **fflush().**

```
# include <stdio.h>
# include <string.h>
# include <conio.h>

# define max 30
struct z
	{
	short edad;
	char nombre[20];
	};

struct z alum[max],aux;
short i,k=0,j;

void main(void)
	{
	clrscr();
	printf("ingrese nombre: ");
	scanf("%s",alum[k].nombre);
	while (strcmp(alum[k].nombre, "salir")!= 0)
		{
		printf("edad:");
		scanf("%d",&alum[k].edad);
		k++;
		printf("ingrese nombre: ");
		scanf("%s",alum[k].nombre);
		}

	for (i=0;i<k-1;i++)
		for (j=i+1;j<k;j++)
```

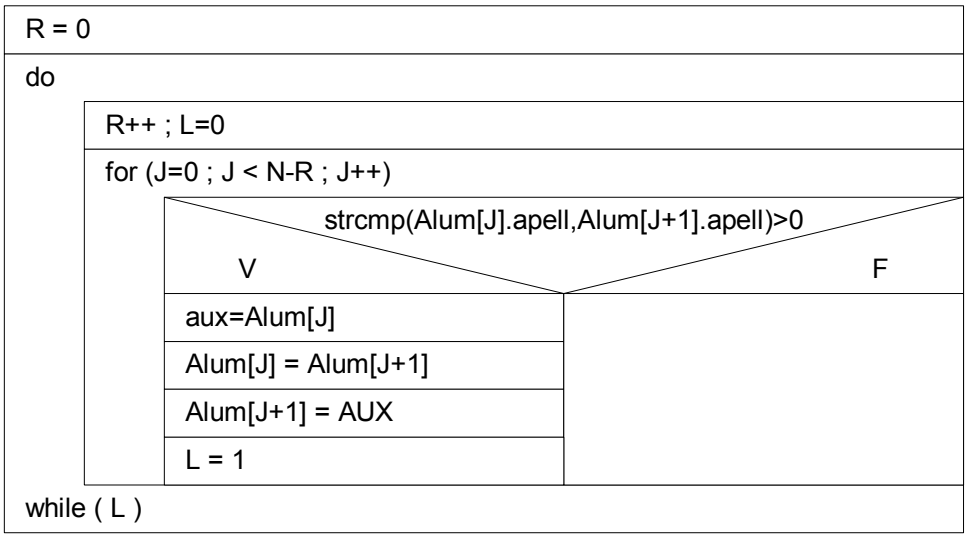
```
                if (strcmp(alum[i].nombre,alum[j].nombre)>0)
                        {
                        aux=alum[i];
                        alum[i]=alum[j];
                        alum[j]=aux;
                        }

        printf("\nOrdenado por nombres\n");
        for (i=0;i<k;i++)
                printf("\nNombre: %20s Edad: %2d",alum[i].nombre, alum[i].edad);
        }
```

Ordenamiento – Múltiples criterios

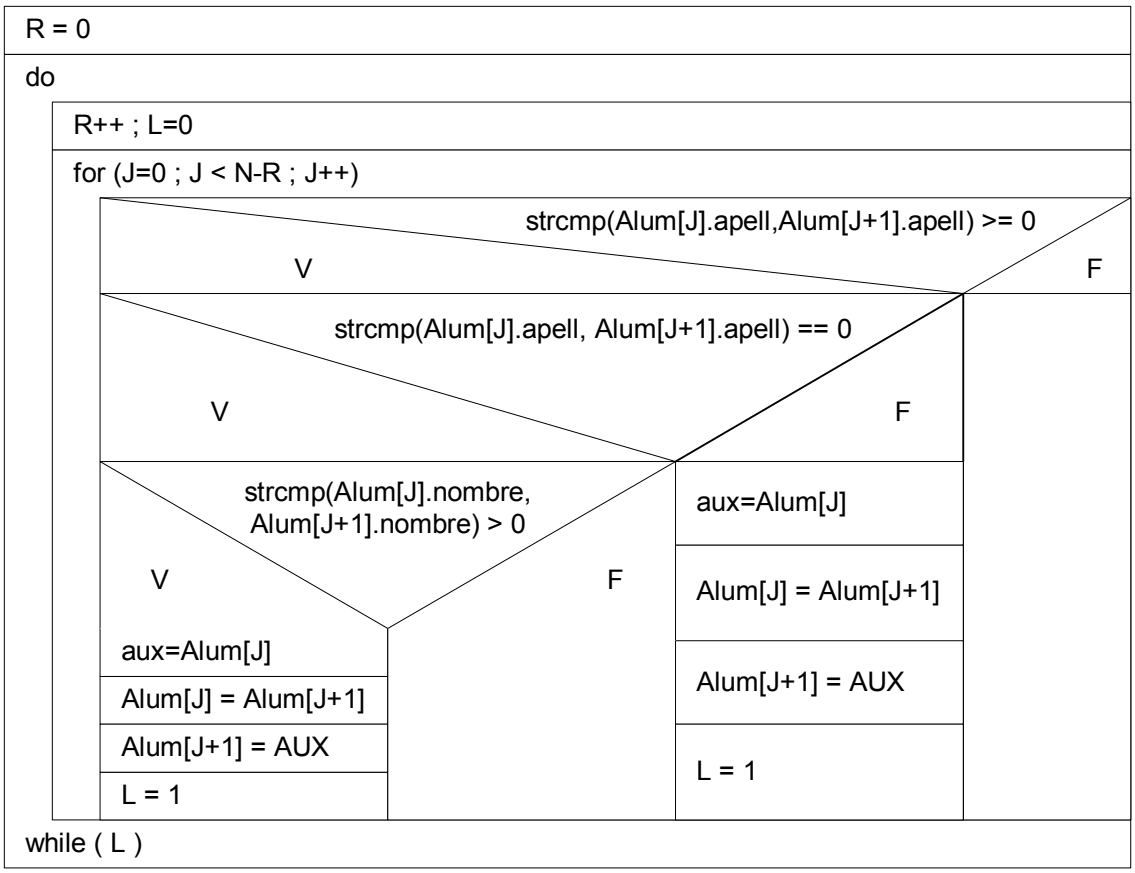
Es frecuente encontrarse con la necesidad de realizar ordenamientos en donde el criterio se basa en más de una variable.
Sea el caso de por ejemplo emitir listados en donde aparezcan los apellidos de los alumnos de una cátedra ordenados alfabéticamente, pero si existen apellidos repetidos debe a su vez ordenarse según el nombre de los mismos. Aqui vemos que el ordenamiento es en base a los apellidos de los alumnos y aparece un **segundo criterio de ordenamiento** -que son los nombres de los alumnos- en caso de alumnos con el mismo apellido. Claro está, el apellido constituye el **primer criterio** por el cual se debe ordenar.
Veamos como poder ir realizando el algoritmo (diagramando el Chapin):
El caso de ordenar según el apellido presenta la siguiente forma (resolvemos el primer criterio):

Estas son las condiciones previas con las que se inicializan las variables

```
# include <stdio.h>
# include <string.h>

# define N 30

struct estudiante
        {
        char apell[21];
        char nombre[11];
        };

struct estudiante alum[N],aux;
short J,R;
```



Ahora bien, en el caso de producirse igualdad de los apellidos se deben ordenar los nombres. Entonces introduzco esta posibilidad dentro del primer condicional y pregunto si **strcmp(Alum[J].apell,Alum[J+1].apell) >= 0** . O sea que si los apellidos están desordenados o son iguales, realizaré un intercambio de los datos. Con este condicional puedo afirmar que si es falso no debo intercambiar, y si es verdadero sí debo intercambiar. Pero dado que no es condición suficiente para permitir el intercambio, deberé introducir un nuevo condicional sólo para el caso en que el primero haya sido verdadero: **strcmp(Alum[J].apell, Alum[J+1].apell)==0**. Ahora, se plantean dos posibilidades, la primera es si el resultado de strcmp() es distinto de cero, con lo cual esta condición es necesaria y suficiente para realizar el intercambio necesario dado que estamos ante el caso de que los apellidos están desordenados. La segunda posibilidad indica que los apellidos son iguales, la cual es condición necesaria pero no suficiente, y hace falta agregar otra condición que se añada y que realice el intercambio si además de cumplirse las condiciones mencionadas, se cumple que los nombres están desordenados. Así es como se introduce **strcmp(Alum[J].nombre, Alum[J+1].nombre) > 0** con el objeto de poder obtener la condición suficiente para realizar el intercambio en el caso que los apellidos sean iguales y los nombres estén desordenados.
Con este razonamiento se obtiene el siguiente diagrama de Chapin que satisface un ordenamiento con doble criterio (el primer criterio es el apellido y el segundo el nombre).



Analicemos por un momento las condiciones en una tabla con todas las combinaciones posibles. Observar que se coloca entre paréntesis el resultado de la función strcmp() del ejemplo dado para mejor entendimiento:

Apellidos	Nombres	Intercambiar ?
Desordenados (>)	Desordenados (>)	Sí
Desordenados (>)	Ordenados (<)	Sí
Ordenados (<)	Desordenados (>)	No
Ordenados (<)	Ordenados (<)	No
Iguales (=)	Desordenados (>)	Sí
Iguales (=)	Ordenados (<)	No

Llevémoslo a otra forma pero suprimiendo aquellas filas que no tienen sentido. Para esto se indica con una X que cualquiera sea el estado de ese criterio el resultado es el mismo:

Primer criterio	Segundo criterio	Intercambiar ?
Desordenados (>)	X	Sí
Ordenados (<)	X	No
Iguales (=)	Desordenados (>)	Sí
Iguales (=)	Ordenados (<)	No

Entonces si consideramos al intercambio que debemos hacer dentro del vector como una función **Intercambiar**, que sea la que indique si hay que realizar intercambio, esta función resultará de los diversos estados que pueden tomar los criterios de ordenamiento. Así, podemos llegar a la siguiente conclusión:

La función **Intercambiar** será **Sí** cuando se cumplan **cualquiera** de estos dos casos:

- Primer criterio indique un estado desordenado.
- Primer criterio indique igualdad y segundo criterio tenga un estado desordenado.

El énfasis en la palabra cualquiera del párrafo anterior es para indicar que los dos casos son cada uno de ellos condición necesaria y suficiente para realizar el intercambio. Es decir, si el primer criterio tiene un estado desordenado se debe intercambiar y no hace falta comprobar más nada. Por otra parte e independiente del caso anterior, si el primer criterio indica igualdad y segundo criterio tiene un estado desordenado se debe intercambiar.

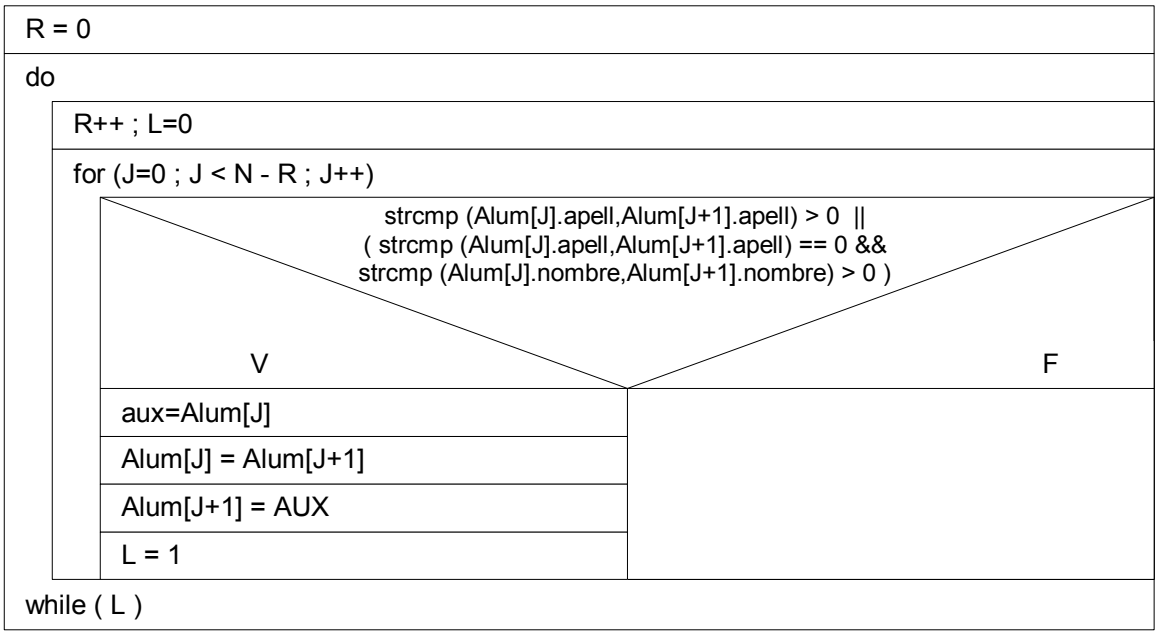
En lógica booleana la forma de aunar los dos casos mencionados más arriba es mediante la utilización de la función OR (suma lógica). Además, para el segundo caso la función lógica booleana que relaciona es la AND (producto lógico) dado que tiene que producirse la simultaneidad para que este caso sea condición necesaria y suficiente. Veámos cómo queda:

(1er criterio desordenado) **OR** (1er criterio con igualdad **AND** 2do criterio desordenado)

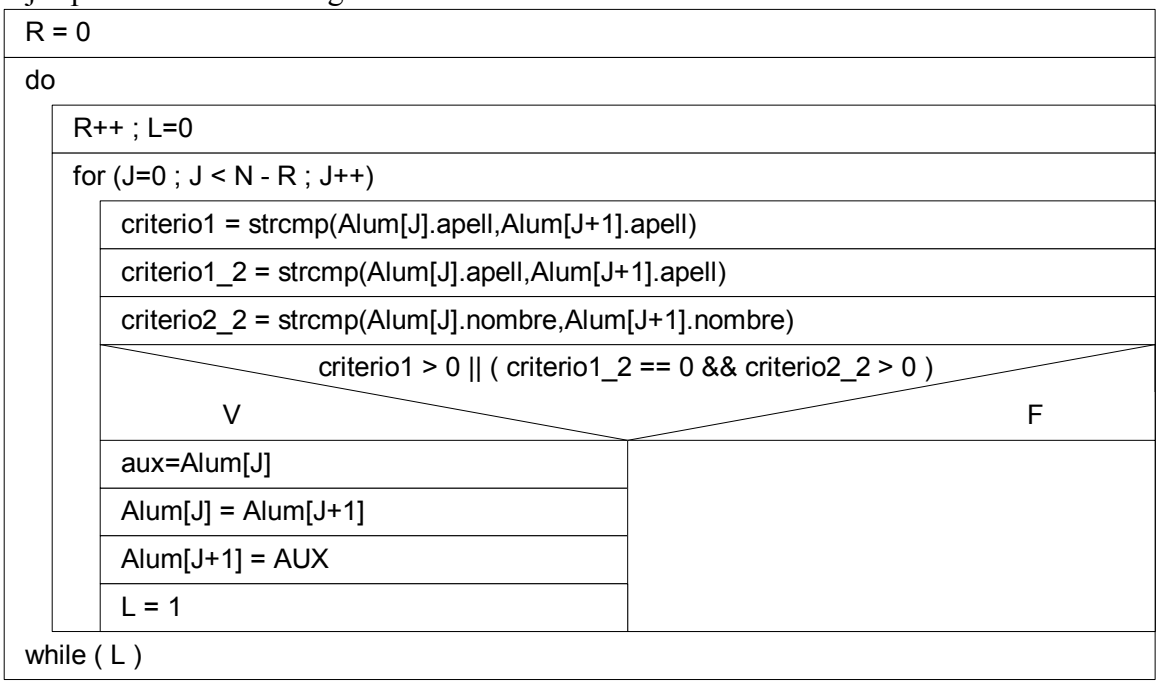
N O T A

Los paréntesis son absolutamente necesarios.

Entonces es fácil ver que podemos resolver un problema con doble criterio de ordenamiento con un solo condicional en el que la condición sea el resultado de una función lógica:



una mejor presentación es la siguiente:



I M P O R T A N T E

En el lenguaje C el nivel de precedencia de la función AND es mayor que el de OR por lo que la codificación en C no requerirá la utilización de los paréntesis. En el diagrama de Chapin los seguimos incluyendo para mayor claridad.

Paso de miembros de una estructura a una función

En la llamada a una función podemos pasarle parámetros -Ver CAPÍTULO 6, Invocación y llamada por valor- para que esta pueda tener la información de dicho parámetro y así realizar algún tipo de operación.

Por ejemplo:
Es el caso en el que se invoca a una función para que calcule el tamaño del archivo en Kbytes:

```
.
.
.
struct info_archivo
       {
       char nombre[9];
       char extension[4];
       char fecha[9];
       short tamanio;
       short archivo_solo_lectura;
       short archivo_oculto;
       short archivo_sistema;
       } archivo;
.
.
.
void main(void)
.
.
.
convierte_a_Kbytes(short);
.
.
.
```

donde la función podría ser:

```
void convierte_a_Kbytes(short length)
       {
       float length_Kbytes;

     length_Kbytes=(float)length/1024;
     printf("\n%.2f", length_Kbytes);
     }
```

N O T A

Se ha utilizado el operador cast debido a que tanto lenght como 1024 son enteros y se desea tener un resultado float

---

Paso de una estructura a una función

También podremos pasar como parámetro toda una estructura a una función.
En el pasaje de una estructura a una función, la información que aquélla posee es pasada ***por valor*** (Ver CAPÍTULO 6, Invocación y llamada por valor). De la misma forma se puede retornar una estructura ya que la misma se considera como un solo valor.

Programa ejemplo,

```
#include <stdio.h>

struct alumno
       {
```

```
       short legajo;
       short edad;
       short p1;
       short p2;
       float promedio;
       };

struct alumno calcula (struct alumno);

void main (void)
     {
     struct alumno datos;
     printf("\nIngrese el legajo del alumno ");
     scanf("%d", &alumnado.legajo);
     printf("\nIngrese edad ");
     scanf("%d", &alumnado.edad);
     printf("\nIngrese nota 1er. parcial ");
     scanf("%d", &alumnado.p1);
     printf("\nIngrese nota 2do. parcial ");
     scanf("%d", &alumnado.p2);
     datos = calcula(datos);
     printf("Legajo\tEdad\tPromedio\n");
     printf("%d\t%d\t%f\n", datos.legajo, datos.edad, datos.promedio);
     }

struct alumno calcula (struct alumno prom)
     {
     prom.promedio = ((float)prom.p1 + prom.p2)/2;
     return (prom);
     }
```

Vemos que se pasó la estructura ***datos*** como parámetro a la función calcula. Dicho pasaje se realiza por valor y reiteamos que debe tenerse en cuenta de que coincidan los tipos de estructuras tanto en la invocación de la función como así también en la definición de la misma, y los ámbitos de validez de las estructuras.
Luego, dentro de la función se actualiza el promedio del alumno calculándolo y guardándolo en el miembro ***promedio*** de la estructura **prom** y luego esta es retornada y guardada en la estructura **datos**.

Ejemplo de aplicación

Veamos ahora la siguiente codificación que utiliza una función con la finalidad de ordenar alfabéticamente según el nombre del archivo

```
#include<stdio.h>
#define cantidad 100

struct info_fecha
       {
       short dia;
       short mes;
       short anio;
       };

struct info_archivo
       {
       char nombre[9];
       char extension[4];
       struct info_fecha fecha;
       short tamanio;
       };
```

```
void ordena (struct info_archivo ARCHI[]);

void main(void)
      {
      struct info_archivo ARCHIVOS[cantidad];
      short indice;
      clrscr();
      for(indice=0;indice<cantidad;indice++)
       {
       printf("Ingrese nombre ");
       fflush(stdin);
       gets(ARCHIVOS[indice].nombre);
       printf("Ingrese extension ");
       fflush(stdin);
       gets(ARCHIVOS[indice].extension);
       printf("Ingrese tamaño ");
       scanf("%d",&ARCHIVOS[indice].tamanio);
       printf("Ingrese día de creación ");
       scanf("%d",&ARCHIVOS[indice].fecha.dia);
       printf("Ingrese mes de creación ");
       scanf("%d",&ARCHIVOS[indice].fecha.mes);
       printf("Ingrese año de creación ");
       scanf("%d",&ARCHIVOS[indice].fecha.anio);
       }

      ordena(ARCHIVOS);

      for(indice=0;indice<cantidad;indice++)
       {
             printf("\nArchivo: %8s.%3s\tTamaño: %d\tFecha: %02d/%02d/%04d",
             ARCHIVOS[indice].nombre, ARCHIVOS[indice].extension,
      ARCHIVOS[indice].tamanio, ARCHIVOS[indice].fecha.dia, ARCHIVOS[indice].fecha.mes,
             ARCHIVOS[indice].fecha.anio);
       }
      }

void ordena (struct info_archivo ARCHI[])
{
short i,j;
struct info_archivo aux;

for (i=0;i<cantidad-1;i++)
       for (j=i+1;j<cantidad;j++)
              if (strcmp(ARCHI[i].nombre,ARCHI[j].nombre)>0)
                     {
                     aux=ARCHI[i];
                     ARCHI[i]=ARCHI[j];
                     ARCHI[j]=aux;
                     }
}
```

En este ejemplo no necesitamos retornar ningún parámetro ya que se pasó a la función la dirección de comienzo del vector ARCHIVOS.

Ejemplo:

De un instituto militar egresan 200 soldados los cuales han sido calificados en 12 diferentes disciplinas (Por ejemplo explosivos, artillería, inteligencia, etc.). Estas disciplinas se codifican del 1 al 12.

Realizar un programa que ingrese los datos de cada uno de los soldados los cuales consisten en número de legajo de cada comando y sus correspondientes doce calificaciones.

A continuación el programa debe interrogar al operador acerca de si desea asignar una misión o no.

En caso de responder afirmativamente el programa debe solicitar la cantidad de comandos requeridos y la disciplina de mayor interés.

Con estos datos se debe emitir un listado de los soldados asignados eligiendo los mejores de la categoría solicitada. Una vez que un comando es asignado para una misión no puede serlo en otra.

Si la cantidad requerida no puede ser satisfecha el programa deberá asignar los soldados disponibles y se deberá emitir una leyenda que así lo indique.

Luego de cumplirse con el informe correspondiente a una misión se deberá interrogar nuevamente al operador acerca de si se asigna o no una misión. En caso de contestar si se debe repetir el procedimiento caso contrario terminar el programa.

En el siguiente problema nos encontramos luego de leer el enunciado con lo siguiente:
Si en lugar de utilizar una estructura con un miembro vector utilizáramos vectores independientes necesitaríamos doce rutinas diferentes de ordenamiento; una solución válida para evitar esto sería usar una matriz para las notas ya que el proceso de ordenamiento sería uno solo pero el arrastre es realmente muy engorroso por lo cual inferimos que la solución propuesta es más dinámica.

```
#include <stdio.h>
#include <conio.h>

void main (void)
{

struct soldado
       {
       short numero,asig;
       float cate[13];
       };

struct soldado comando[200],aux;

short i,j,categ,cant,F,r,cupo;
float au;
char resp;

clrscr();
for(i=0;i<200;i++)
       {
       printf("\n ingrese numero:");
       scanf("%d",&comando[i].numero);
       comando[i].asig=1;

              for(j=1;j<13;j++)
              {
              printf("\t ingrese nota categoria %d:",j);
              scanf("%f",&au);
              comando[i].cate[j]=au;
              }

       }

printf("desea asignar una mision s/n:");
resp=getche();
while(resp=='s'|| resp=='S')
       {
       printf("\n ingrese categoria:");
       scanf("%d",&categ);
       printf("\n ingrese cantidad:");
       scanf("%d",&cant);
       cupo=0;
       r=0;
```

```
                do
                for(r++,i=0,F=0;i<200-r;i++)
                if(comando[i].cate[categ]<comando[i+1].cate[categ])
                        {
                        aux=comando[i];
                        comando[i]=comando[i+1];
                        comando[i+1]=aux;
                        F=1;
                        }
                while(F);
        for(i=0;i<200 && cupo<cant ; i++)
        if(comando[i].asig)
                {
                printf("\n comando numero:%d\t\t",comando[i].numero);
                printf("calificacion:%f",comando[i].cate[categ]);
                comando[i].asig=0;
                cupo++;
                }
        if(cupo<cant)
        printf("\n No hay mas soldados");
        printf("\n Desea aignar una mision s/n:");
        resp=getche();
        }
}
```

I M P O R T A N T E

Podemos notar que las notas no son cargadas directamente sino que se utiliza una variable auxiliar au; esto es debido a problemas que se presentan el ejecutar el programa y que no son alertados por el compilador o sea estamos ante la presencia de un bug que se presenta reiteradamente cuando tenemos miembros float, double o long double en un vector de estructuras.

Uniones

Las uniones son un tipo de estructura en la cual todos sus miembros comparten el espacio en memoria ocupado por el miembro mas grande. A diferencia de la estructura, el tamaño total que ocupa esta en memoria es la suma del tamaño que ocupa cada uno de sus miembros.

La forma general de definición es

```
union  nombre_unión
       {
       campos de la unión
       };
```

I M P O R T A N T E

No se debe olvidar el ';' después de cerrar llaves '}'.

Ejemplo

```
union _GRUPO
      {
      char nombre[10];
      char inicial;
      };
```

Creamos una unión y sus elementos son un nombre de 10 bytes (nombre[10]) y la inicial (1 byte). Como hemos dicho la unión ocupa el espacio de su elemento más grande, en este caso nombre. Por lo tanto la unión ocupa 10 bytes. Las variables nombre e inicial comparten el mismo sitio de la memoria. Si accedemos a nombre estaremos accediendo a los primeros 10 bytes de la unión (es decir, a toda la unión), si accedemos a inicial lo que tendremos es el primer byte de la unión.

```
#include <stdio.h>

union _GRUPO
      {
      char nombre[10];
      char inicial;
      } pers;

int main()
     {
     printf("Escribi tu nombre: ");
     gets(pers.nombre);
     printf("\nTu nombre es: %s\n", pers.nombre);
     printf("Tu inicial es: %c\n", pers.inicial);
     }
```

Ejecutando el programa:

Escribi tu nombre: pepe

Tu nombre es: pepe
Tu inicial es: p
Para comprender mejor eso de que comparten el mismo espacio en memoria vamos a ampliar el ejemplo. Si añadimos unas líneas al final que modifiquen sólo la inicial e imprima el nuevo nombre:

```
#include <stdio.h>
```

```
union _GRUPO
      {
      char nombre[10];
      char inicial;
      } pers;

int main()
     {
     printf("Escribi tu nombre: ");
     gets(pers.nombre);
     printf("\nTu nombre es: %s\n", pers.nombre);
     printf("Tu inicial es: %c\n", pers.inicial);
     /* Cambiamos la inicial */
     pers.inicial='Z';
     printf("\nAhora tu nombre es: %s\n", pers.nombre);
     printf("y tu inicial es: %c\n", pers.inicial);
     }
```

Tendremos el siguiente resultado:
Escribe tu nombre: pepe

Tu nombre es: pepe
Tu inicial es: p

Ahora tu nombre es: carlos
y tu inicial es: c
Aquí queda claro que al cambiar el valor de la inicial estamos cambiando también el nombre porque la inicial y la primera letra del nombre son la misma posición de la memoria.
Con las uniones podemos usar punteros de manera similar a lo que vimos en el capítulo de las estructuras.
Las uniones son especialmente útiles ya que permiten almacenar información de diferentes tipos dentro de los registros. En programación de sistemas es usual encontrarlas dentro de las estructuras de datos de las rutinas, ya que permiten una gran flexibilidad a la hora de almacenar información.

Campos de bits

Hay un nuevo tipo de datos que solo se puede usar con estructuras: el campo de bits. Un campo de bits es un elemento de una estructura que se define en términos de bits, y se comporta igual que un entero sin signo, sólo que al definir el campo de bits se define el número de bits que lo compondrá .

La declaración de un campo de bits requiere que se escriba el nombre del tipo de dato, la variable y seguido de dos puntos, el ancho del campo en bits de ese miembro.

Citemos dos ejemplos:

```
struct teclado
       {
      unsigned char
              rshift : 1;
       lshift : 1;
              ctrl          : 1;
       alt           : 1;
              scroll : 1;
       numlock       : 1;
              caplock       : 1;
       insert : 1;
       };
```

```
struct comida
     {
     unsigned clase : 2; /* dos bites para el tipo */
     unsigned temporada : 1; /* a 1 si es de temporada */
     unsigned es_perecedero :1, es_congelado : 1;
     };
```

Si el tamaño del campo de bits es 0 nos permite alinear el siguiente campo sobre un entero. Hay que tener en cuenta que la alineación de los campos de bits y de los demás campos la define el compilador.

Hay que tener en cuenta que no se puede obtener la dirección de un campo de bits. El uso de campos de bits permite empaquetar información pequeña eficientemente dentro de una estructura. Su uso está bastante extendido en la programación de sistemas.

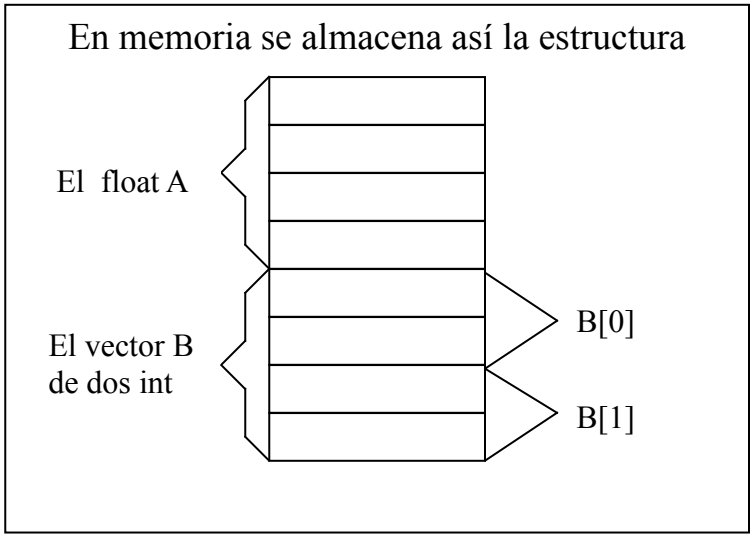
Alternativa al tema Union y campo de bits

Uniones y campos de bits

Una unión no es ni mas ni menos que lo mismo que una estructura a nivel de declaración y manejo. El único “cambio” entre la unión y la estructura es a nivel de almacenamiento en memoria. Esto se ejemplifica a continuación:

Imaginemos la siguiente definición de una estructura:

```
struct datos{
        float A;
        int B[2];
};
```

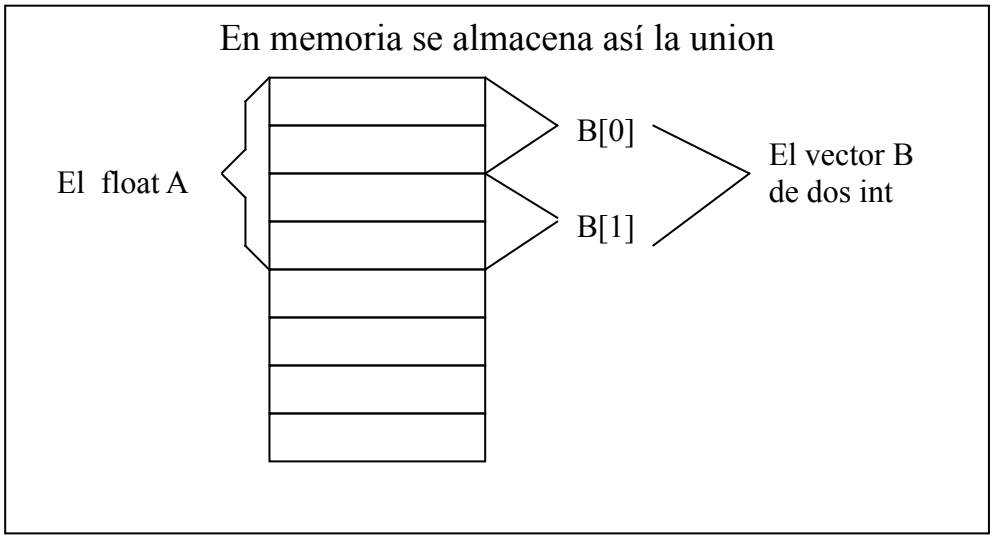


Vamos a ver como definimos un tipo de dato union y luego como se almacena en la memoria (dato que es de vital importancia conocer muy bien).

```
union datos{
        float A;
        int B[2];
    };
```

Como se vé lo único que cambia es que hemos reemplazado la palabra reservada struct por la palabra reservada union.

¿Cómo se almacenan los datos en memoria? Bueno, como sigue en el gráfico:



Como se observa ahora, los datos estarán superpuestos en memoria a partir de la dirección cero de la zona de memoria donde se encuentre almacenada la union. ¿Y entonces como hago para acceder al float o al vector de int? ¿Para qué me sirve esto? Te sirve para poder tener distintas vistas de lo mismo. Yo puedo ver al mismo conjunto de 4 bytes como un único dato float o como dos datos enteros que forman un vector (¿Por qué un vector de enteros y no dos enteros?). Si a la union le hubiese agregado un vector de cuatro char podría haber visto ahora a la misma región como cuatro bytes separados (muy útil como veremos, para transmitir datos por medio de puertos).
¿Cómo uso un dato de tipo union? Se utiliza exactamente igual que si fuera una estructura.

Siguiendo el mismo ejemplo de la union

```
union datos{
        float A;
        int B[2];
    };

void main (void)
{
```

```
        union datos info;

        scanf("%f, &info.A);
        prinft("%f, info.A);

        printf("%d", info.B[0]);
        printf("%d", info.B[1]);
}
```

¿Qué veríamos si ejecutamos el programa?
Veríamos el valor float ingresado, y dos valores enteros que serian muy extraños en la mayoría de los casos. Es que lo que estamos viendo es el valor decimal de los dos bytes almacenados en memoria que forman parte del formato de almacenamiento del byte. También podríamos ver el valor decimal de los cuatro bytes que forman parte del almacenamiento del float en memoria si incluyéramos en la declaración de la union a un vector de cuatro char. Este formato de almacenamiento del float corresponde a la norma IEEE que se estudiará en detalle en digitales I e informática II.

En los próximos ejemplos se verá con mas claridad el uso de las uniones, para ello nos va a servir definir previamente que es un campo de bits.

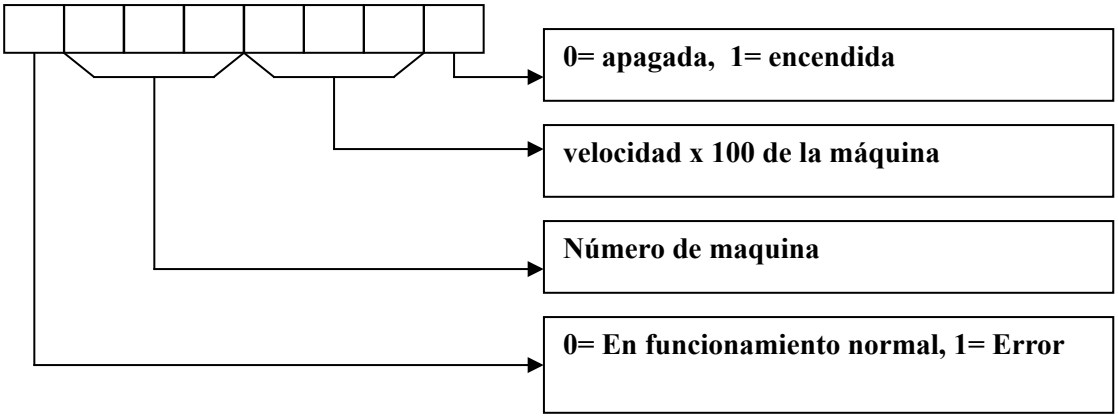
Definición de campo de bits

Un campo de bits es una estructura que toma la siguiente forma particular:

```
        struct mis_bits{
                                unsigned char bit0_al_2: 3;
                                unsigned char bit3: 1;
                                unsigned char bit4_y_5:2;
                                unsigned char bit6_ y_7:2;
                        };
```

Lo primero que vemos, es que hemos creado un tipo de dato llamado struct mis_bits. Después observamos que cada campo se define en forma similar a lo que hacíamos antes con las estructuras (¡si es una estructura, como va a ser diferente!!!!!) salvo por la inclusión después de cada campo de un dos puntos seguido de un número. Primero y fundamental, lo que estamos haciendo con toda esta estructura, es definir bits o conjuntos de bits de un byte o conjunto de bytes. El dos puntos y el numero representan los bits de ese byte que vamos a definir y a los que les damos un indentificador determinado. Por ejemplo, en nuestro ejemplo le estamos dando el nombre de bit0_al_2 a los 3 bits más bajos de nuestro byte.

Imaginemos la siguiente situación: por medio de nuestro puerto serie recibimos un byte desde una serie de maquinas. Este byte contiene información del estado y funcionamiento de cada máquina junto con el nombre de la misma, tal como se especifica a continuación:



Vamos a crear el campo de bits para esta situación en particular:

```
struct mis_bits{
        unsigned char encendida: 1;
        unsigned char velocidad: 3;
        unsigned char maquina: 3;
        unsigned char error: 1;
};
```

Ahora bien, nosotros dijimos que la información nos llegaba a través del puerto serie, esto es un byte. Por lo tanto tendremos que "superponer"nuestro campo de bits a un byte, entonces usamos una union entre nuestro campo de bits y un byte.

```
union todos{
        struct mis_bits informacion;
        unsigned char dato_recibido;
    };
```

Ahora en el main creamos un dato de tipo union todos

```
union todos mis_datos;
```

ahora voy a recibir el byte desde una de las máquinas:

```
mis_datos.dato_recibido = inportb(0X3F8);
```

A continuación voy a chequear si hay error y en caso de haberlo informo el numero de maquina

```
if (mis_datos.informacion.error = = 1)
printf("La maquina %u ha reportado un error",mis_datos.información.maquina);
```

Si quisiera imprimir la velocidad tendría que hacer: mis_datos.información.velocidad*100.

Cuando nos referimos como en este caso de la velocidad a un cojunto de dos o mas bits el valor lo podemos manejar en decimal o en hexadecimal según más nos guste y convenga. Si imprimiéramos la velocidad los posibles valores que veriamos serian: 0, 100, 200, 300, 400, 500, 600 y 700, y si imprimiéramos el numero de máquina, iría de 0 a 7. Supongamos que pudiéramos modificar desde

nuestro software la velocidad de la maquina, tendríamos que enviar como byte el numero de maquina (por ejemplo 5), la velocidad (por ejemplo 200), el error en 0 o en 1 (no importa, ya que no lo podemos modificar nosotros porque es información que nos envia la maquina) y el encendida en 1. Procederemos de la siguiente forma:

```
mis_datos.informacion.maquina = 5;
mis_datos.informacion.velocidad = 2;
mis_datos.informacion.encendida = 1;
mis_datos.información.error = 0;
outportb(0X3F8, mis_datos.dato_recibido);
```

Esto mismo se pudo haber hecho con una mascara usando operadores a nivel de bit. Cada uno elegirá el método que le sea más claro y conveniente.

No es necesario cada vez que se desea modificar un conjunto en particular de bits acceder a todos. Con modificar el que queremos alcanza, ya que el resto permanece invariable (lo mismo que hicimos con operadores a nivel de bits, esto es: leo el byte desde el puerto, le modifico lo que quiero, y para finalizar lo coloco donde estaba).

Nota: inportb y outportb son dos funciones de Borland de la dos.h. La función es la de ingresar y sacar datos de tipo char (1 byte a la vez) por medio de puertos. Podrán ser reemplazadas para las pruebas por printf y scanf o por las funciones de manejo de puertos que cada entorno y compilador implemente.

Técnicas de acceso a dispositivos ASCII y “no ASCII”

Vamos a comenzar en verdad por lo que debería ser el resumen final, para luego comenzar la explicación de cada punto en particular.

Tipo de dispositivo	Método a utilizar
Binario	uniones
ASCII	sscanf, ssprintf y otras funciones que convierten valores numéricos enteros y reales a cadenas de caracteres (strings) y viceversa.
Binario y ASCII	uniones como método más optimo (¡No el único método!). Aquí puedo elegir.

Cuando decimos dispositivo, nos estamos refiriéndo a por ejemplo otra PC (de la cual recibo o le envío datos) o un cartel al que le envío un texto (como los que hay en muchas inmobiliarias, colectivos, estaciones de trenes y subtes). Como ya hemos visto, solo puedo sacar por un puerto bytes y de a uno a la vez. Entonces ¿Qué hago si por ejemplo tengo que mandar un float, que ni siquiera es un entero y para colmo ocupa mas de un byte de memoria y con un protocolo de almacenamiento complejo?. Bueno, para esto van a existir varias técnicas que nos van a permitir “desarmar” el dato para adaptarlo al port de salida (y en función de lo que es capaz de manejar el dispositivo destino) y luego “reconstruirlo” en el dispositivo de destino.

Vamos a ver funciones y técnicas que nos permitirán ese “armar” y “desarmar” del dato.

Las funciones sscanf y sprintf

Estas dos funciones de la familia del scanf y printf nos permitirán manejarnos con cadenas ASCII (string). Permitirán por ejemplo desarmar en caracteres ASCII al float de nuestro ejemplo. Recordemos que la función prinft convierte al dato numérico a cadena de caracteres y que la pantalla de la PC no es ni mas menos que una matriz de caracteres. En cambio, al teclado lo podríamos llegar a ver como una fuente de caracteres ASCII, que después la función scanf “convierte” al formato numérico correspondiente.

Imaginemos que tenemos el siguiente float -123456.789 al que se lo convierte en el string de caracteres ‘-’’1’’2’’3’’4’’5’’6’’.’’7’’8’’9’ es decir que el flotante que como “número” ocupa 4 bytes de memoria, como string ocupará una cantidad variable de bytes (en este caso 11) y en general cantidad bastante mayor a 4 bytes. Ahora bien, ¿de que forma logramos convertir el numero al string? Veamos el siguiente ejemplo que descompone un float a su equivalente string y se envía por el puerto serie:

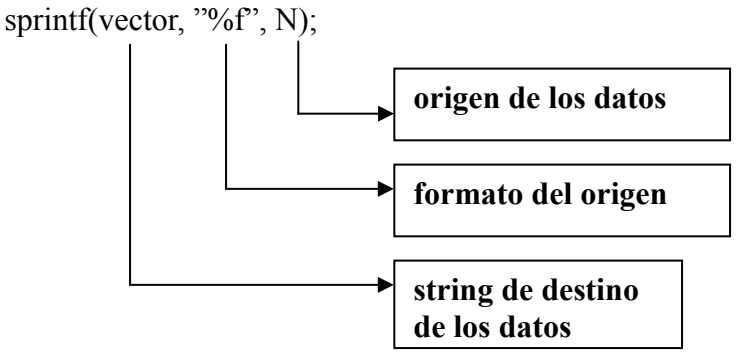
```
#include<stdio.h>
#include<conio.h>

void main(void)
{
      float N = -123456.789;
      char vector[12];
                        int i;

//armo el vector de caracteres
sprintf(vector, ”%f”, N);

//envío los datos byte a byte
for(i=0;i<12;i++)
outportb(0X3F8, vector[i]);
}
```

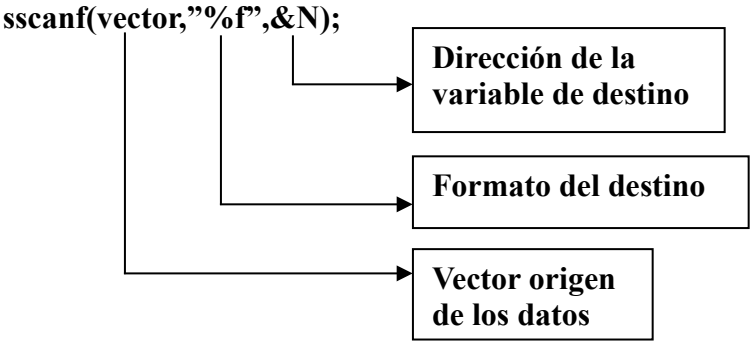
Para completar el ejemplo veremos como se recibe la información y se reconstruye el float original.

```
#include<stdio.h>
#include<conio.h>

void main(void)
{
      float N = -123456.789;
      char vector[12];
                                  int i;

     //recibo los datos desde el puerto
      for(i = 0; i < 12; i++)
                                    vector[i]=inportb(0X3F8);
```

```
//"reconstruyo" el dato
                    sscanf(vector,"%f", &N);

//lo imprimo como numero en la pantalla
                    printf("El valor recibido es: %f", N);
                }
```

En estos ejemplos hemos utilizado sprintf y sscanf. sprintf nos ha permitido convertir el float a una cadena string





En resumen sscanf en lugar de recibir los datos desde teclado como lo hace scanf los recibe desde un vector de caracteres y sprintf en lugar de enviar los datos a "pantalla" como printf los envía a un vector de caracteres. El manejo de sscanf y sprintf es idéntico al de scanf y printf, salvo lo anteriormente indicado. Se deberá tener especial cuidado en el dimensionado del vector contando todos los caracteres, incluyendo, espacios, punto decimal y signo además de no olvidarse que hay que dejar espacio para el carácter NULL.

Uso de uniones como método para "armar" y "desarmar" datos

Otra forma de descomponer un dato, es aprovechar la característica que tiene la union entre distintos tipos de datos para ver la misma información de distintas formas. Vuelvo a re-cordar lo mismo que dije anteriormente: por un puerto no podemos transmitir (o recibir) otra cosa que no sean bytes. Por lo tanto, al dato que sea (int, float, double, etc), deberemos descomponerlo en N bytes que después será reconstruido en el receptor. Como regla general, si en el transmisor uso uniones en el receptor

también uso uniones para reconstruir el dato.
En el ejemplo que sigue voy a ver al mismo dato como un float o como un vector de 4 bytes según me convenga en cada momento.

Intentemos de nuevo enviar y recibir el mismo float del ejemplo anterior.

```
#include<stdio.h>
#include<conio.h>

union datos{
        float real;
    unsigned char byte[4];
        };

void main(void)
{
    int i;
    union datos info;

//"cargo" el campo real
    info.real=-123456.789;

//ahora envío por el port byte a byte
for(i=0; i<4; i++)
        outportb(0X3F8,info.byte[i]);
}
```

Vamos a recibir del "otro lado" la información que acabamos de transmitir.

```
#include<stdio.h>
#include<conio.h>

union datos{
        float real;
    unsigned char byte[4];
        };

void main(void)
{
```

```
                    int i;
                    union datos info;

//recibo byte a byte
                    for(i=0; i < 4; i++)
                        info.byte[i]=inportb(0X3F8);

                    //imprimo en pantalla el dato recibido
                    printf("El dato que se recibió es: %f", info.real);
                }
```

¿Si por lo que se puede ver, utilizar union es mas efectivo (envío (recibo) 4 bytes en lugar de 12 bytes), por qué ulilizar la técnica del vector de string? La respuesta es muy sencilla: ¡NO SIEMPRE SE PUEDE!!! Se observa que enviamos 3 veces menos datos que con el vector de string ganando en efectividad en la transmisión de la información, mas si la conexión es lenta y si el volumen a transmitir es grande. Si tengo que transmitir información de una PC a otra lo mas conveniente será utilizar uniones. Si es un cartel que recibe datos ASCII tendré que utilizar en forma obligatoria sscanf y sprintf u otras funciones que hagan lo mismo (ver las notas complementarias).